INSTITUT POLYTECHNIQUE DE PARIS

5OD14 — Lecture Notes, AA 2024 / 2025

Cooperative optimization and learning

Lecture notes, version at November 26, 2024

Andrea Simonetto

Forewords

Context and Aim

This course covers the theory and the algorithms to solve cooperative optimization and learning problems. Cooperative problems arise naturally in a number of recent applications, such as distributed computing, massive-scale machine learning, and the Internet of Things (IoT). In all these application scenarios, the data and therefore the optimization and learning cost and loss are distributed in space across multiple devices. Due to communication and privacy issues, we cannot gather all the data at a single location, and we are "forced" to look for alternative, advanced, algorithms that can solve optimization and learning problems in a cooperative fashion.

In particular, **the aim of the course is to be able to answer the questions,**

1. What is a cooperative optimization and learning problem?
2. Given a cooperative problem, which algorithm do I use to solve it? And with which theoretical guarantees?
3. How do I ensure privacy in the algorithms we develop and what do we mean by privacy?

In order to answer to these three questions, we will need to build a theory of cooperative algorithms. This will make us discover some very recent development in optimization and learning, such as the ADMM algorithm, federated learning, as well as differential privacy. Some of the algorithms we will study are implemented in some way or another by the big players in the field (Google, Microsoft, Meta, ...), and run on your browsers and on your smart phones.

The notes and course give for granted a good knowledge of continuous optimization and algorithms, for example the content of 4OPT1 and 4OPT2 at ENSTA.

Note. The sections or subsections marked with $*$ contain optional advanced material, which is not covered in class. Research papers are papers which you will study and report in groups.

References. The theory of cooperative optimization and learning is developing at a fast pace. I will refer to books and papers as we go along. Good and up-to-date introductions on the topic can be found in,

- [BPC+11] : S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*, Foundations and Trend® in Machine Learning, 2011
- [NNC19] : G. Notarstefano, I. Notarnicola, A. Camisa, *Distributed optimization for smart cyber-physical networks*, Foundations and Trends® in Systems and Control, 2019
- [KM+21] : P. Kairouz, B. H. McMahan, *et al.*, *Advances and open problems in federated learning*, Foundations and Trends® in Machine Learning, 2021

Palaiseau, November 26, 2024.

Contents

Chapter 1

Convex optimization and cooperation problems

1.1 Introduction

In this course, we will look at optimization problems that involve cooperations among a set of players, or agents and algorithms that are tailored to solve them. This is vastly motivated by current trend in machine learning and the Internet of Things (IoT), where data collection is distributed among many devices and we want to process it locally instead of gathering it at a centralized location. While all these terms and description may be vague at this point, it will make sense after we describe formally the elements of our study.

We start by revisiting generic optimization problems of the form

$$\begin{aligned} (\mathsf{P0}) \quad \underset{\boldsymbol{x}\in X\subseteq\mathbf{R}^n}{\text{minimize}} \quad & f(\boldsymbol{x}) && (1.1)\\ \text{subject to} \quad & g(\boldsymbol{x}) \leqslant 0 && (1.2)\\ & h(\boldsymbol{x}) = 0, && (1.3)\end{aligned}$$

where the functions $f(\boldsymbol{x}) : \mathbf{R}^n \to \mathbf{R}$, $g(\boldsymbol{x}) : \mathbf{R}^n \to \mathbf{R}^p$, $h(\boldsymbol{x}) : \mathbf{R}^n \to \mathbf{R}^q$. In addition, X represents a generic closed set. As it appears, we look at continuous optimization problems for which $\boldsymbol{x} \in \mathbf{R}^n$. We indicate with $\boldsymbol{x}^*$ a (global) optimizer of the original problem, and by $f^* = f(\boldsymbol{x}^*)$ the unique (global) minimum. We also indicate the Euclidean inner product as $\langle v, u\rangle = v^\top u = u^\top v$ for two vectors $u, v \in \mathbf{R}^n$.

We will mainly consider **convex** problems, for which X is a convex set, $g(\boldsymbol{x})$ is a convex function and $h(\boldsymbol{x})$ is affine. I will give for acquired most of the optimization theory you have seen in the past two years (in particular, in the course OPT201 and OPT202).

Said so, the problem we will look at in this course has the form,

$$(\mathsf{P}) \qquad \min_{\boldsymbol{x}\in X\subseteq\mathbf{R}^n} \sum_{i=1}^N f_i(\boldsymbol{x}), \tag{1.4}$$

where the set X is closed and convex, $f_i : \mathbf{R}^n \to \mathbf{R}$ is a convex function for all i's and the minimum is attained for a $\boldsymbol{x} \in X$. This is a simpler version of ($\mathsf{P0}$) with a very special structure: the cost is a sum of N different costs. We can imagine to have multiple (N) devices that want to "agree" on an optimal decision $\boldsymbol{x}$, trading off their individual costs f_i's.

Before moving on, I want to further motivate the setting with a few interesting examples.

Example 1.1 Cooperative least-squares. *Imagine N users collect noisy measurements $\boldsymbol{y}_i \in \mathbf{R}^n$ about a quantity $\boldsymbol{x} \in X$. A way to estimate the true value for $\boldsymbol{x}$ is to set up a least-squares problem as*

$$\min_{\boldsymbol{x}\in X} \sum_{i=1}^N \|\boldsymbol{x} - \boldsymbol{y}_i\|^2 = \sum_{i=1}^N f_i(\boldsymbol{x})$$

Example 1.2 Cooperative linear model training. *Imagine N users collect input-output pairs $\boldsymbol{w}_i \in \mathbf{R}^{n-1}, \boldsymbol{y}_i \in \mathbf{R}$ (e.g., features and labels), and they want to train a global model with weights $\boldsymbol{x} \in X \subseteq \mathbf{R}^n, \boldsymbol{x} = [\theta \in \mathbf{R}^{n-1}, c \in \mathbf{R}]$. Let the local input-output mapping be affine,*

$$\boldsymbol{y}_i = \theta^\top \boldsymbol{w}_i + c, \qquad \forall i,$$

then the training problem can be written as

$$\min_{\boldsymbol{x}\in X} \sum_{i=1}^N \|\boldsymbol{y}_i - (\theta^\top \boldsymbol{w}_i + c)\|^2 = \sum_{i=1}^N f_i(\boldsymbol{x})$$

Example 1.3 Cooperative network problems. *Take a sensor network of N sensors. You want to compute the localization of the whole network based on pair-wise distance measurements (e.g., sensors can be cars, or people with their phones). Then each measurement is*

$$m_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\| + \textit{noise}, \qquad \boldsymbol{x}_i \in \mathbf{R}^2 \textit{ is the position of node } i.$$

You have E pair-wise measurements. Then you can write the problem as

$$\min_{\boldsymbol{x}\in X\subset \mathbf{R}^{2E}} \sum_{i,j}^E (m_{ij} - \|\boldsymbol{x}_i - \boldsymbol{x}_j\|)^2 = \sum_{k=1}^E f_k(\boldsymbol{x})$$

Example 1.4 Cooperative consensus. *You have N robots at different locations $\boldsymbol{s}_i$ and moving at different speeds v_i, and you want to find the best position in space for the fastest rendezvous:*

$$\min_{\boldsymbol{x}\in X\subset \mathbf{R}^3} \sum_i^N \frac{\|\boldsymbol{s}_i - \boldsymbol{x}\|}{v_i} = \sum_{i=1}^N f_i(\boldsymbol{x})$$

These are just a few examples, we will see more complex ones as we continue in the course. So, what are the main challenges of the course? After all we should know how to solve (P) from previous courses. The main challenges are that,

- Despite the fact that the local devices "want to" cooperate to solve the problem, the local cost function f_i may be built on private data that cannot be shared (think about private messages);
- The local functions f_i may be built on data whose size is big enough not to be practical to share and gather (think about video streaming);
- The local devices have limited communication capabilities and they can only communicate their computations to a selected number of other devices – and they do not have access to a cloud system.

These challenges imply that we need to solve part of (P) on the local devices and then send the local computations to other devices in order to cooperate and reach a *consensus* on what the optimal variable should be.

1.2 Preliminaries

1.2.1 Devices and network

Let's introduce some definitions.

Definition 1.1 (Device) *We call device, agent, or player, an entity which is embedded with (possibly limited) computation and communication capabilities. Devices are endowed with a cost function $f_i : \mathbf{R}^n \to \mathbf{R}$, and possibly local constraints $\boldsymbol{x} \in X_i \subseteq \mathbf{R}^n$ built upon the data they collect.*

Definition 1.2 (Communication network) *A device can communicate with other devices. We define the set of communication links, the communication network.*

Devices and communication network will play the central role in our algorithms to solve (P). Given the limited computation capabilities of the devices, we will mainly look at first-order algorithms.

1.2.2 Convexity

The problems we will look at will be convex in most part of the course. Let us recap some useful notions and recall that a function is $\mathcal{C}^n(X)$ iff it is continuously differentiable up to degree n on the domain $X \subseteq \mathbf{R}^n$.

Definition 1.3 (Convex functions) *A function $f : X \subseteq \mathbf{R}^n \to \mathbf{R}$ is convex iff X is convex and*

$$\textit{(C1)} \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X, \lambda \in [0,1]: \quad f(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) \leqslant \lambda f(\boldsymbol{x}) + (1-\lambda) f(\boldsymbol{y}).$$

Multiple definitions exists, for example:

$$\begin{aligned} (\mathsf{C1}) + f \in \mathcal{C}^1(X) \quad &\Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ f(\boldsymbol{x}) \geqslant f(\boldsymbol{y}) + \langle \nabla f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \rangle \\ (\mathsf{C1}) + f \in \mathcal{C}^2(X) \quad &\Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ \nabla^2 f(\boldsymbol{x}) \succeq 0 \end{aligned}$$

Definition 1.4 (Strongly convex functions) *A convex function $f : X \subseteq \mathbf{R}^n \to \mathbf{R}$ is m-strongly convex iff*

$$\textit{(SC)} \quad f(\boldsymbol{x}) - \frac{m}{2}\|\boldsymbol{x}\|^2 \textit{ is convex.}$$

It is important to note that a strongly convex function f does not need to be differentiable. Multiple definitions exists, for example:

$$\begin{aligned} (\mathsf{SC}) + f \in \mathcal{C}^1(X) \quad &\Longleftrightarrow \\ &\forall \boldsymbol{x}, \boldsymbol{y} \in X,\ \langle \nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \rangle \geqslant m\|\boldsymbol{x} - \boldsymbol{y}\|^2 \\ (\mathsf{SC}) + f \in \mathcal{C}^2(X) \quad &\Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ \nabla^2 f(\boldsymbol{x}) \succeq m I_n \end{aligned}$$

Definition 1.5 (Smooth functions) *A convex function $f : X \subseteq \mathbf{R}^n \to \mathbf{R}$ is L-smooth iff*

$$\textit{(LC)} \quad \frac{L}{2}\|\boldsymbol{x}\|^2 - f(\boldsymbol{x}) \textit{ is convex.}$$

Here, it is important to remember that (LC) implies differentiability, i.e., $f \in \mathcal{C}^1(X)$. Multiple definitions exists, for example:

$$(\mathsf{LC}) \quad \Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ \|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\| \leqslant L\|\boldsymbol{x} - \boldsymbol{y}\| \tag{1.5}$$

$$(\mathsf{LC}) \quad \Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ \frac{1}{L}\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\|^2 \leqslant \langle \nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \rangle \tag{1.6}$$

$$(\mathsf{LC}) \quad \Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ f(\boldsymbol{x}) \geqslant f(\boldsymbol{y}) + \langle \nabla f(\boldsymbol{y}), \boldsymbol{x} - \boldsymbol{y} \rangle + \frac{1}{2L}\|\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})\|^2 \tag{1.7}$$

$$(\mathsf{LC}) + f \in \mathcal{C}^2(X) \quad \Longleftrightarrow \quad \forall \boldsymbol{x}, \boldsymbol{y} \in X,\ 0 \preceq \nabla^2 f(\boldsymbol{x}) \preceq L I_n \tag{1.8}$$

which can be found in [Nes04].

We now define the set $\mathcal{S}^p_{m,L}(\mathbf{R}^n)$ as the set of functions defined over $\mathbf{R}^n$, that are p-differentiable, m-strongly convex, and L-smooth. In this course, we will consider mainly functions that are at least in $\mathcal{S}^1_{m,L}(\mathbf{R}^n)$, with $0 < m \leqslant L < +\infty$.

1.2.3 Gradient algorithm

To finish with the recap, we will revise the vanilla gradient algorithm, which is still the most important first-order algorithm to solve optimization problems of the form of (P).

Consider solving the convex problem,

$$\underset{\boldsymbol{x} \in X \subseteq \mathbf{R}^n}{\text{minimize}}\, f(\boldsymbol{x}) \tag{1.9}$$

with $f : \mathbf{R}^n \to \mathbf{R}$ convex, and X a closed convex set. Recall that $\mathsf{P}_X[\cdot]$ is the Euclidean projection onto the set X, i.e.,

$$\mathsf{P}_X[v] = \arg\min_{\boldsymbol{x}\in X} \frac{1}{2}\|\boldsymbol{x} - v\|^2. \tag{1.10}$$

Then, the projected gradient method is the following algorithm.

Projected gradient method

- *Start with* $\boldsymbol{x}_0 \in \mathbf{R}^n$ *and a positive stepsize sequence* $\{\alpha_k\}_{k\in\mathbb{N}}$
- *Iterate* $\boldsymbol{x}_{k+1} = \mathsf{P}_X[\boldsymbol{x}_k - \alpha_k \nabla f(\boldsymbol{x}_k)], \quad k = 0, 1, \ldots.$

There are different methods to choose the stepsize sequence α_k, either a priori or in an online fashion. For example:

- We can take it constant: $\alpha_k = \alpha > 0$
- We can take it vanishing: $\alpha_k = \frac{\alpha}{\sqrt{k+1}} > 0$
- We can select the best α_k that ensures the biggest decrement, etc.

The different choices will dictate the convergence properties of the algorithm. Let us recap one useful result.

Theorem 1.1 *The projected gradient method on Problem (1.9) with constant stepsize $\alpha < 2/L$, for functions $f \in \mathcal{S}^1_{m,L}$ has a (global) convergence guarantee of*

$$\|\boldsymbol{x}_t - \boldsymbol{x}^*\| \leqslant \rho^t \|\boldsymbol{x}_0 - \boldsymbol{x}^*\| \tag{1.11}$$

for $\rho = \max\{|1 - \alpha m|, |1 - \alpha L|\} < 1$, and t iterations.

Proof. *[Sketch] Start with the update rule: $\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| = \|\mathsf{P}_X[\boldsymbol{x}_k - \alpha\nabla f(\boldsymbol{x}_k)] - \boldsymbol{x}^*\|$. Recall that $\boldsymbol{x}^* = \mathsf{P}_X[\boldsymbol{x}^* - \alpha\nabla f(\boldsymbol{x}^*)]$ and that the projection is a non-expansive operator, meaning,*

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| = \|\mathsf{P}_X[\boldsymbol{x}_k - \alpha\nabla f(\boldsymbol{x}_k)] - \boldsymbol{x}^*\| \leqslant \|\boldsymbol{x}_k - \boldsymbol{x}^* - \alpha(\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}^*))\|.$$

If $f \in \mathcal{S}^1_{m,L}$, then the function $x \mapsto x - \alpha\nabla f(x)$ is contraction, with contraction parameter $\rho = \max\{|1 - \alpha m|, |1 - \alpha L|\}$ (Homework: how do we prove this?), as such,

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| \leqslant \rho\|\boldsymbol{x}_k - \boldsymbol{x}^*\|,$$

selecting $\alpha < 2/L$, then $\rho < 1$, and we can iterate backwards the previous relation and prove the theorem. ♣

We call this convergence *linear*, and we recall that linear convergence is the best we can achieve for first-order algorithms in this functional class.

1.3 A first cooperation problem and algorithms

We are now ready to setup our first cooperation problem. Imagine we want to solve,

$$\operatorname*{minimize}_{\boldsymbol{x}\in\mathbf{R}^n} \sum_{i=1}^N f_i(\boldsymbol{x}), \tag{1.12}$$

and we are in the setting of Figure 1.1, where "you" can communicate with all the devices and the devices can communicate with you. The devices are first-order oracles, i.e., they can evaluate the function values and compute its gradient at a testing point. What can you do to solve the problem?

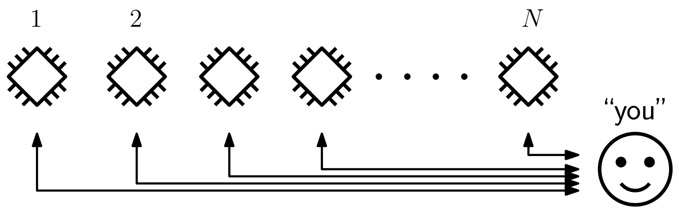


Figure 1.1. *The first cooperation problem with N devices endowed with their local cost function f_i that can be queried by "you".*

1.3.1 Sequential approach

The most natural way of proceeding is a sequential approach: you ask one device to apply a local gradient, then you get back the result, then you ask a second device, and so forth. This sequential approach has been given different names along the years. It is mostly known under the name of Gauss-Seidel gradient or incremental gradient, and it forms the basis of stochastic gradient descent in machine learning (as we will see later). In machine learning, it is sometimes referred to as online back-propagation or finite sum stochastic gradient descent.

Incremental/ Gauss-Seidel Gradient

- *Start with* $\boldsymbol{x}_0 \in \mathbf{R}^n$ *and a positive sequence* $\{\alpha_k\}_{k\in\mathbb{N}}$
- *Iterate: pick a device* i *and query it:* $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f_i(\boldsymbol{x}_k), \quad k = 0, 1, \ldots.$

We can use this simple algorithm to describe a few properties and characteristics we will look for in our collaborative algorithms.

For the incremental method, we can say that,

- **Privacy.** The data that generates f_i stays private, since the devices only communicate the updated $\boldsymbol{x}_{k+1}$ and not the gradient nor the function.
- **Communication robustness.** The algorithm is robust to communication issues, such as package drops, delays: you ask a device to compute its update and you can wait for it as long as you want. If you don't receive the update, you can pass to the next device, but the algorithm is not affected by it.

These two properties will be important in all the methods we will see, so we will come back to it every time.

Let us now characterize its convergence guarantees, i.e., is the incremental gradient able to converge to the optimizer of the problem $\boldsymbol{x}^*$?

Theorem 1.2 *Consider Problem (1.12) for a m-strongly convex and L-smooth function $f(\boldsymbol{x}) := \sum_{i=1}^N f_i(\boldsymbol{x})$. Consider the incremental gradient method with a constant stepsize $\alpha < 2/L$. Assume that*

$$\|\nabla f_i(\boldsymbol{x}) - \sum_{i=1}^N \nabla f_i(\boldsymbol{x})\| \leqslant G, \qquad \forall\, \boldsymbol{x} \in \mathbf{R}^n\ .$$

Define the contraction rate, $\rho = \max\{|1-\alpha m|, |1-\alpha L|\} < 1$.

Then convergence of the incremental gradient goes as

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| \leqslant \rho\|\boldsymbol{x}_k - \boldsymbol{x}^*\| + \alpha G.$$

Theorem 1.3 *Under the same conditions of Theorem 1.2, if the stepsize is chosen vanishing as $\alpha_k = 1/k^s$, for a constant $0 < s < 1$ then, we obtain the convergence guarantee:*

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\| \leqslant O(1/k^s).$$

Proof. *We are going to prove only Theorem 1.2. For the function $f(\boldsymbol{x}) = \sum_{i=1}^N f_i(\boldsymbol{x})$. Write*

$$\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\| \leqslant \|\boldsymbol{x}_k - \alpha\nabla f(\boldsymbol{x}_k) - \boldsymbol{x}^* + \alpha\nabla f(\boldsymbol{x}^*)\| + \alpha\|\nabla f_i(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}_k)\|.$$

Use now the strongly convex and smoothness property to say (as in the gradient method's proof):

$$\|\boldsymbol{x}_k - \alpha\nabla f(\boldsymbol{x}_k) - \boldsymbol{x}^* + \alpha\nabla f(\boldsymbol{x}^*)\| \leqslant \rho\|\boldsymbol{x}_k - \boldsymbol{x}^*\|,$$

from which the thesis follows.

The proof of Theorem 1.3 is a bit more involved but not impossible, see for example [GOP19]. ♣

From the theorems, we can see already that, for incremental gradient, we either have linear convergence to an error bound as,

$$\limsup_{k\to\infty} \|\boldsymbol{x}_k - \boldsymbol{x}^*\| = \frac{\alpha G}{1-\rho}, \tag{1.13}$$

or slower (i.e., $O(1/k^s)$) convergence to the exact optimizer $\boldsymbol{x}^*$. This would be a recurrent aspect of cooperative algorithms.

As for the assumption: the bound G measures how different is the local function from the true function. A lot of research has been devoted to lifting the assumptions (strong convexity, smoothness), and relaxing the assumption on G, but the basic ideas are still valid: you have a trade-off between speed and error.

1.3.2 A more complete incremental gradient characterization*

Research paper 1 *Björn Johansson, Maben Rabi, and Mikael Johansson,* A Randomized Incremental Subgradient Method for Distributed Optimization in Networked Systems*, SIAM Journal on Optimization, Vol. 20 (3), 2010*

1.3.3 Surpassing the gradient method in large-scale applications*

Research paper 2 *Aryan Mokhtari, Mert Gürbüzbalaban, and Alejandro Ribeiro,* Surpassing Gradient Descent Provably: A Cyclic Incremental Method with Linear Convergence Rate*, SIAM Journal on Optimization, Vol. 28 (2), 2018*

1.3.4 Parallel approach

A second, very natural way of proceeding (if you have the bandwidth to ask or a cloud service) is that you ask every device to apply a local gradient, then you get back the result and perform some global computation. We won't get too much into the type of global computation you can do for the moment, since it will depend on the situation, but the general algorithm looks as follows.

Parallel/ Jacobi Gradient

- *Start with* $\boldsymbol{x}_0 \in \mathbf{R}^n$ *and a positive sequence* $\{\alpha_k\}_{k\in\mathbb{N}}$
- *For all* $k = 1, 2, \dots$ *iterate:*
  - *Ask every devices:* $\boldsymbol{x}^i_{k+1} = \boldsymbol{x}_k - \alpha_k\nabla f_i(\boldsymbol{x}_k), \quad \forall i$
  - *Compute* $\boldsymbol{x}_{k+1}$ *based on the local updates* $\boldsymbol{x}^i_{k+1}$

The algorithm is known under the name of parallel or Jacobi gradient and will form the basis of much of distributed optimization and federated learning. The scheme remains private, in the sense that the devices communicate only the update $\boldsymbol{x}^i_{k+1}$, but not the function nor the gradient, but the scheme is less robust to communication issues: we need to wait all the answer of all the devices before updating our scheme.

1.4 Communication network

Before moving on to more complex algorithms and the true content of the course, the last thing that we need revising is a bit of graph theory to formalize the concept of communication network.

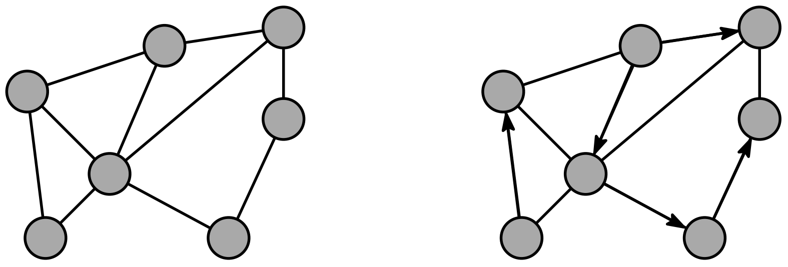


Figure 1.2. .

Consider a graph $\mathcal{G}$ as a collection of vertices $\mathcal{V}$, edges $\mathcal{E}$, and edge weights $\mathcal{W}$. We define **undirected** a graph for which an edge can carry information in both directions. Otherwise we call it **directed**. In this sense, if the devices can send and receive information from another device, the communication link is undirected, otherwise it is directed. If all the links are undirected, then the communication network (or equivalently, graph) is undirected, otherwise directed. Figure 1.2 depicts our graphical conventions in terms of undirected and directed graphs.

If two devices can communicate, we say that they share an edge of the graph. We define the **Adjacency matrix** as $\mathcal{A} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, with 1 entries if device i and j share an edge.

1.4.1 Undirected graphs

For undirected graphs, we define **the vertex degree** d^i as the number of neighbors or edges vertex i has. Note that different authors have different notations: for us, in the undirected case, vertex i is not a neighbor of itself, so if i is not connected to any other vertex, its $d^i = 0$.

We also define the **Laplacian matrix** as an $|\mathcal{V}| \times |\mathcal{V}|$ symmetric matrix defined as

$$\mathcal{L}_{\mathcal{G}} = \mathcal{D} - \mathcal{A}, \tag{1.14}$$

where $\mathcal{D}$ =diag$(d^1, \ldots, d^n)$ is the degree matrix, which is the diagonal matrix formed from the vertex degrees. From the definition, it is easy to show that $\mathcal{L}_{\mathcal{G}}\mathbf{1} = \mathbf{0}$, and therefore the vector $\mathbf{1}$ is a right eigenvector of the Laplacian with eigenvalue 0.

We define with N_i the set of nodes that node i communicates to (its neighborhood). We say that the graph is connected if every node can communicate to any other node in within a number of communication rounds, or equivalently, if there is a path between each vertex of the graph.

In Table 1.1, we report a few examples of undirected communication graphs and their Laplacian matrix.

To study the convergence of cooperative algorithms, the following class of matrices is important enough to have its own definition:

Definition 1.6 (Doubly stochastic matrix) *A doubly stochastic matrix W is a matrix for which, each entry is non-negative, and $W\mathbf{1} = \mathbf{1}$, and $\mathbf{1}^\top W = \mathbf{1}^\top$.*

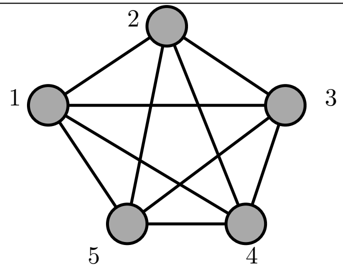
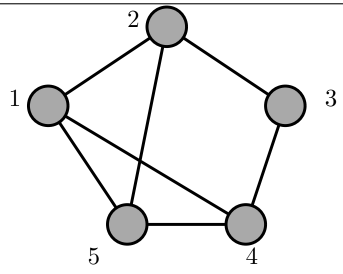
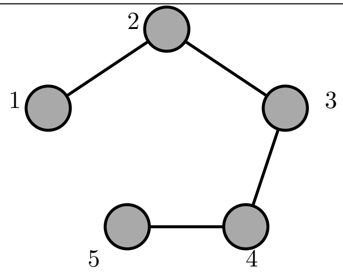
Doubly stochastic matrices will be defined as weight matrices on graphs. Examples of this construction are the following important ones.

Example 1.5 (Metropolis and Lazy Metropolis weights) *Consider a undirected and connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. The Metropolis-Hastings weight matrix is the matrix defined as,*

$$W^{ij} = \begin{cases} \frac{1}{1+\max\{d^i,d^j\}} & \text{for } (i,j) \in \mathcal{E} \\ 1 - \sum_j W^{ij} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases} .$$

Here d^i is the degree (rem: the number of neighbors) of node i, and W^{ij} is the i,j entry of the matrix $W \in \mathbf{R}^{|\mathcal{V}|\times|\mathcal{V}|}$. Such a W is doubly stochastic.

Table 1.1. *Examples of undirected communication graphs*

Graph example	Name	$\mathcal{L}_\mathcal{G}$
	full graph	$\mathcal{L}_\mathcal{G} = \begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{bmatrix}$
	a generic graph	$\mathcal{L}_\mathcal{G} = \begin{bmatrix} 3 & -1 & 0 & -1 & -1 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 2 & -1 & 0 \\ -1 & 0 & -1 & 3 & -1 \\ -1 & -1 & 0 & -1 & 3 \end{bmatrix}$
	line graph	$\mathcal{L}_\mathcal{G} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}$

The Lazy Metropolis weight matrix is the matrix defined as,

$$W^{ij} = \begin{cases} \frac{1}{2(1+\max\{d^i,d^j\})} & \text{for } (i,j) \in \mathcal{E} \\ 1 - \sum_j W^{ij} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}.$$

Such a W is also doubly stochastic and it has a number of interesting properties (as we will see next).

Example 1.6 (Laplacian weights) *Consider a undirected and connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and its Laplacian matrix $\mathcal{L}_\mathcal{G}$. Then then weight matrix,*

$$W = I - \varepsilon \mathcal{L}_\mathcal{G},$$

with $\varepsilon < \max_i\{d^i\}$ is a doubly stochastic matrix.

Once a communication matrix W is defined, we can compute its eigenvalues, which are important for the convergence properties of a given algorithm. For that, we can use the Gershgorin circle theorem.

Theorem 1.4 (Gershgorin circle theorem) *Let A be a complex $n \times n$ matrix, with entries a_{ij}. For $i \in \{1, \ldots, n\}$ let R_i be the sum of the absolute values of the non-diagonal entries in the i-th row:*

$$R_i = \sum_{j \neq i} |a_{ij}| .$$

Let $D(a_{ii}, R_i) \subseteq \mathbb{C}$ be a closed disc centered at a_{ii} with radius R_i. Such a disc is called a Gershgorin disc.

Then, every eigenvalue of A lies within at least one of the Gershgorin discs $D(a_{ii}, R_i)$.

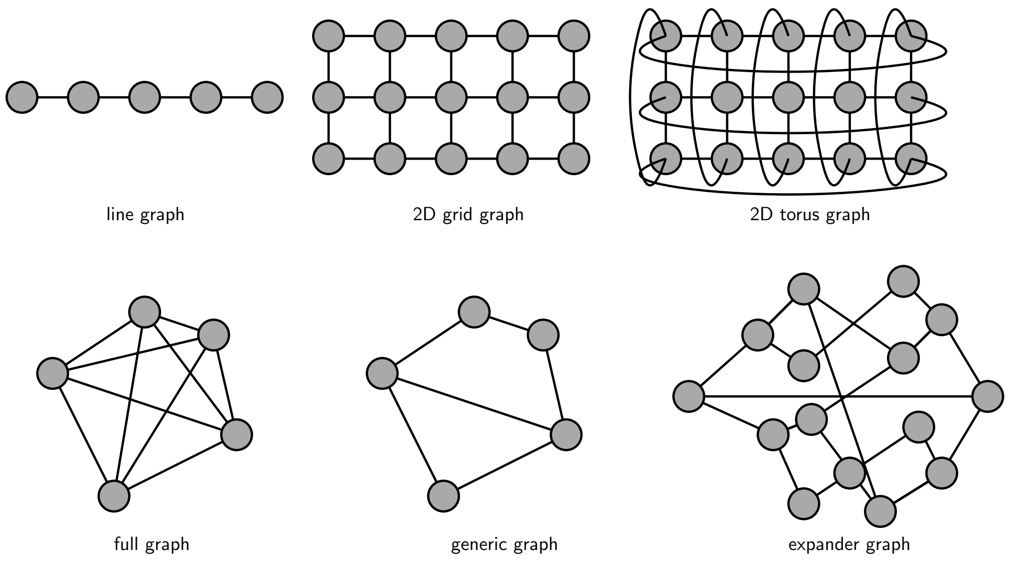


Figure 1.3. *Example of notable graph structures.*

Proof. *See [HJ12, Theorem 6.1.1]* ♣

By using the theorem, for doubly stochastic matrices of dimension $N \times N$, we can order its eigenvalues as,

$$1 = \lambda_1(W) \geqslant \lambda_2(W) \geqslant \cdots \geqslant \lambda_N(W) \geqslant -1,$$

define the second largest eigenvalue in modulus as,

$$\gamma := \max_i \{|\lambda_2(W)|, |\lambda_N(W)|\},$$

and define the spectral gap as: $\tau := \frac{1}{1-\gamma}$. This quantity is important for convergence: as we will see, the smallest this quantity is, the better the convergence. Consider Figure 1.3. We have the following result.

Theorem 1.5 (Spectral gap) *For the Lazy Metropolis weight matrix of dimensions $N \times N$, the spectral gap can be bounded as follows,*

1. *Line graphs: $\tau = O(N^2)$;*
2. *2-dimensional grid: $\tau = O(N \log(N))$;*
3. *2-dimensional torus: $\tau = O(N)$;*
4. *Full graphs: $\tau = O(1)$;*
5. *Generic graphs: $\tau = O(N^2)$*
6. *Expander graphs: $\tau = O(1)$;*
7. *Erdös-Rényi random graphs (with high probability): $\tau = O(1)$.*

Proof. *The proof uses the concept of hitting times in Markov chains, see [NOR18].* ♣

The theorem loosely says that the more connected a undirected graph is, the better it is for communication. Different results can be found for different doubly stochastic matrices W, but the underlying ideas and trends remain similar.

1.4.2 Directed graphs

Directed graphs are graphs with directed communication links, so some device can communicate with some other device, but not the other way around. Directed graphs are harder to characterize and they have some non-intuitive properties, such as the more communication links do not always guarantee the better convergence.

For each vertex i, we define the out-degree $d^i_{\rm out}$, as the number of neighbors it can communicate to. We also define the in-degree $d^i_{\rm in}$, as the number of vertices they communicate to it (i.e., the number of incoming arrows).

A directed graph is strongly connected if there exists a directed path from any initial vertex to every other vertex in the graph.

1.4.3 Types of communication networks and issues

Finally, in this course, we will look at two main communication settings. The first is the Cloud-based setting, where each device can communicate to a master server, or the cloud. The second is the peer-to-peer setting, where each device can communicate with other devices, but no device is more important nor communicate with everybody.

We will further consider issues such as: asynchronous communication and packet-losses. The first issue arises when devices are not synchronized. The second arises when the communication messages (or packet) are lost before being delivered.

1.5 Examples

In the course, we will look at some interesting machine learning examples, which we describe next.

1.5.1 Kernel ridge regression

Kernel ridge regression is a machine learning task that amounts to fit a functional model to noisy data, in a non-parametric form. You can think of it as linear regression plus plus.

Let y_i for $i = 1, \dots, n$ be scalar evaluations of a certain unknown function $f(x) : \mathbf{R} \to \mathbf{R}$ at points x_i for $i = 1, \dots, n$. In our example, we let the evaluations be noisy measurements as,

$$y_i = f(x_i) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{1.15}$$

with noise ϵ drawn from a normal distribution with mean 0 and variance σ^2.

We now use a kernel representation of the function as

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i), \tag{1.16}$$

where $k(x, x_i)$ is the kernel. Here we will use an Euclidean kernel as

$$k(x, x_i) = \exp(-\|x - x_i\|^2). \tag{1.17}$$

We also define the kernel matrix as $K = [k(x_i, x_j)]_{i,j=1,\dots,n}$.

Function f is linear in the parameters α_i. To lower the computational requirements, we also use a Nyström approximation: that is we select uniformly at random $m \sim \sqrt{n}$ points, say x'_j, for $j = 1, \dots, m$, not necessarily amongst the x_i already described.

The approximation then reads

$$f(x) = \sum_{i=1}^{n} \alpha_i k(x, x_i) \approx \sum_{j=1}^{m} \alpha_j k(x, x'_j). \tag{1.18}$$

Determining the vector $\alpha \in \mathbf{R}^m$ is a model training problem which can be written as,

$$\alpha^* = \arg\min_{\alpha \in \mathbf{R}^m} \left\{ \frac{\sigma^2}{2} \alpha^\top K_{mm} \alpha + \frac{1}{2} \|y - K_{nm}\alpha\|_2^2 \right\}, \tag{1.19}$$

where $K_{nm} = [k(x_i, x'_j)]_{i=1,\dots,n; j=1,\dots,m}$, and y is the stacked version of all y_i.

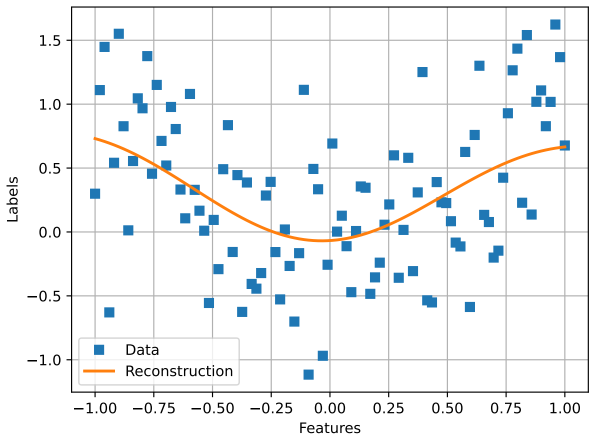


Figure 1.4. *An example of kernel regression problem in one dimension.*

Note then that,

$$\|y - K_{nm}\alpha\|_2^2 = \sum_{i=1}^{n} \|y_i - [k(x_i, x'_j)]_{j=1,\dots,m}\alpha\|_2^2. \tag{1.20}$$

To make things simpler, we also add a small regularization, such that,

$$\alpha^* = \arg\min_{\alpha \in \mathbf{R}^m} \left\{ \frac{\sigma^2}{2}\alpha^\top K_{mm}\alpha + \frac{1}{2}\|y - K_{nm}\alpha\|_2^2 + \frac{\nu}{2}\|\alpha\|_2^2 \right\}, \tag{1.21}$$

for $\nu > 0$, and the problem becomes strongly convex and smooth.

If all the data is available in one single location, we can solve for α^* via the optimality conditions, and

$$\left[\sigma^2 K_{mm} + K_{nm}^\top K_{nm} + \nu I\right]\alpha^* = K_{nm}^\top y, \tag{1.22}$$

which is a linear algebra problem. We can see in Figure 1.4 an example of result.

Consider now N devices, labeled $a = 1, \dots, N$. Suppose these devices to collectively collect the measurements y_i, $i = 1, \dots, n$, such that each device collects some of the measurements and no device collect the same measurement. We let $\mathcal{Y}_a = \{i | y_i \text{ is collected by device } a\}$. As such,

$$\bigcup_{a=1}^{N} \mathcal{Y}_a = \{1, \dots, n\}, \qquad \bigcap_{a=1}^{N} \mathcal{Y}_a = \varnothing.$$

The problem then reads,

$$\alpha^* = \arg\min_{\alpha \in \mathbf{R}^m} \sum_{a=1}^{N} \left[\frac{\sigma^2}{N}\frac{1}{2}\alpha^\top K_{mm}\alpha + \frac{1}{2}\sum_{i \in \mathcal{Y}_a} \|y_i - K_{(i)m}\alpha\|_2^2 + \frac{\nu}{2N}\|\alpha\|_2^2 \right] = \sum_{a=1}^{N} f_a(\alpha),$$

where we indicated with $K_{(i)m} = [k(x_i, x'_j)]_{j=1,\dots,m}$. This problem fits our setting and we will show how to solve it with different methods, so that each device can reconstruct the same function, while sharing limited information with its neighbors (and not its measurements).

1.5.2 Classification with Support Vector Machines (SVM)

The following example is taken from [FCT+20].

Suppose there are N devices, where each device i is equipped with m_i points $p_{i,1}, \dots, p_{i,m_i} \in \mathbf{R}^d$ (which represent training samples in a d-dimensional feature space). Moreover, suppose the points are associated to binary labels, that is, each point $p_{i,j}$ is labeled with $\ell_{i,j} \in \{-1, 1\}$, for all $j \in \{1, \dots, m_i\}$ and $i \in \{1, \dots, N\}$. The problem consists of building a classification model from the training samples. In particular, we look for a separating hyperplane of the form

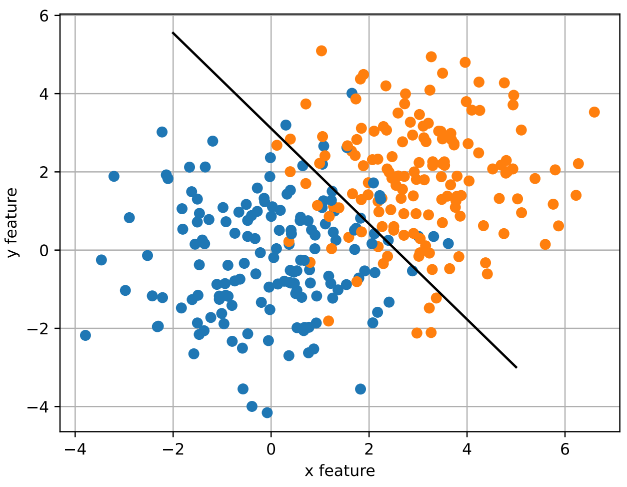


Figure 1.5. *An example of the SVM problem to find the best separating hyperplane in two dimensions. Blue has label +1 and orange −1.*

$\{z \in \mathbf{R}^d | w^\top z + b = 0\}$ such that it separates all the points with $\ell_i = -1$ from all the points with $\ell_i = 1$.

In order to maximize the distance of the separating hyperplane from the training points, one can solve the following (convex) quadratic program:

$$\underset{w\in\mathbf{R},b\in\mathbf{R},\{\xi_{i,j}\in\mathbf{R}\}}{\text{minimize}} \quad \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\sum_{j=1}^{m_i}\xi_{i,j} \tag{1.23}$$

$$\text{subject to} \quad \ell_{i,j}(w^\top p_{i,j} + b) \geqslant 1 - \xi_{i,j} \qquad \forall j, i \tag{1.24}$$

$$\xi_{i,j} \geqslant 0 \qquad \forall j, i, \tag{1.25}$$

where $C > 0$ is a parameter affecting regularization. The optimization problem above is called *soft-margin SVM* since it allows for the presence of outliers by activating the variables $\xi_{i,j}$ in case a separating hyperplane does not exist. Notice that the problem can be viewed as a cooperation problem with cost

$$f(w,b,\xi) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}\sum_{j=1}^{m_i}\xi_{i,j} = \sum_{i=1}^{N}\left(\frac{1}{2N}\|w\|^2 + C\sum_{j=1}^{m_i}\xi_{i,j}\right) = \sum_{i=1}^{N} f_i(w,b,\xi),$$

and local constraint set X_i given by,

$$X_i = \{(w,b,\xi) | \xi \geqslant 0, \ell_{i,j}(w^\top p_{i,j} + b) \geqslant 1 - \xi_{i,j}, \quad \forall j \in \{1,\ldots,m_i\}\}.$$

We can then write the problem as,

$$\underset{w\in\mathbf{R},b\in\mathbf{R},\{\xi_{i,j}\in\mathbf{R}\}}{\text{minimize}} \sum_{i=1}^{N} f_i(w,b,\xi), \quad \text{subject to } (w,b,\xi) \in X_1 \times X_2 \times \cdots X_N. \tag{1.26}$$

The goal is to make devices agree on a common solution (w^*, b^*, ξ^*), so that all of them can compute the soft-margin separating hyperplane as $\{z \in \mathbf{R}^d | (w*)^\top z + b^* = 0\}$.

We will see that this problem is a cost-coupled problem with local constraints.

1.5.3 Multi-class classification with the MNIST dataset

We further consider a multi-class classification task that is standard in machine learning literature: classifying hand-written digits based on the MNIST database (a database of labeled

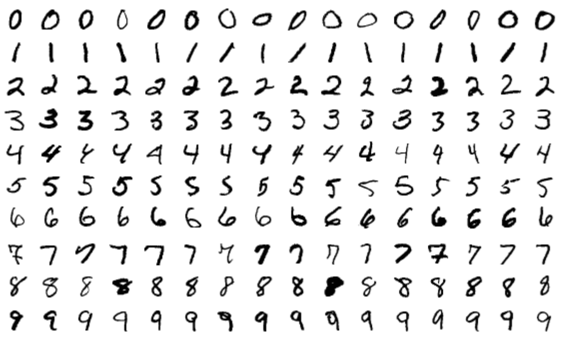


Figure 1.6. *An example of MNIST dataset of hand-written digits.*

hand-written digits). This classification task was very important to read cheques and popularized convolutionary neural networks (CNN) for imaging.

In this example, each digit and label represents a point in your space. An image of a digit, composed of a vector of pixel values is the feature $\boldsymbol{x}_i$, while its numerical value is the label associated with it, y_i. In machine learning, we often define a model, meaning a parametrized function, which should map features to labels, that is we define an $f(\boldsymbol{x};\boldsymbol{w}) : \mathbf{R}^n \times \mathbf{R}^m \to \{0,\dots,9\}$, which should map any image of a digit to its numerical value (between 0 and 9).

We then call the process of finding the best weights $\boldsymbol{w} \in \mathbf{R}^m$ of the function, as training our model. To train the model, we need to choose a model architecture (meaning the shape of f), and a criterion, i.e., a loss that needs minimizing.

We choose the following model and loss.

Multinomial logistic regression. We let $f(\boldsymbol{x};\boldsymbol{w})$ denote the prediction model with the parameter $\boldsymbol{w} = (W, b)$ and the form

$$f(\boldsymbol{x};\boldsymbol{w}) = \mathrm{softmax}(W\boldsymbol{x} + b)$$

where softmax is the function $\mathbf{R}^n \to [0,1]^K$ for K classes ($K = \{0,\dots,9\}$) given per class as,

$$[\mathrm{softmax}(\boldsymbol{z})]_i := \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}.$$

The loss function is instead given by the cross-entropy loss,

$$F(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N} \mathrm{CrossEntropy}[f(\boldsymbol{x}_i;\boldsymbol{w}), y_i] + \lambda\|\boldsymbol{w}\|^2,$$

for N data points, and regularization $\lambda > 0$. The cross-entropy function is (for us) the following function,

$$\mathrm{CrossEntropy}[f(\boldsymbol{x}_i;\boldsymbol{w}), \sigma_i] := \sum_{k=1}^{K} [-\sigma_{ik}\log([f(\boldsymbol{x}_i;\boldsymbol{w})]_k) - (1-\sigma_{ik})\log(1 - [f(\boldsymbol{x}_i;\boldsymbol{w})]_k)]$$

where we have defined $\sigma_{ik} \in [0,1]^K$ as the vector with 1 for the index k corresponding to digit y_i and 0 otherwise.

This is a convex optimization problem. Can you work out why?

We will see in the course how to distribute the database among different devices and train a global model starting from local models.

Chapter 2

Distributed optimization: Primal methods

2.1 First-order algorithms and oracles

We start our journey into cooperative algorithms by looking at primal methods for distributed optimization. We define distributed (sometimes referred to as decentralized) optimization, as the research domain pertaining the study of algorithms to solve coupled optimization problems. For us, this will be all the algorithms that can solve our overarching problem,

$$(\mathsf{P}) \qquad \min_{\boldsymbol{x}\in X\subseteq\mathbf{R}^n} \sum_{i=1}^{N} f_i(\boldsymbol{x}). \tag{2.1}$$

Distributed optimization started in the 1990's with the availability of considerable computer power and distributed computation capabilities, and it has peaked in the 2010's. It is now a quite vibrant research area, with still interesting open problems. The then-standard book in the domain was [BT97] –which still contains tons of useful material–, and we will follow [BPC+11, NNC19], as well as a few research papers, which I will cite as we go along.

First of all, let us recall a few notions that were presented in OPT202.

We start by defining the main players of our theory. For a given optimization problem, we denote its class as $\mathcal{F}$. The class is the collection of properties that that problem has. For example, $\mathcal{F}$ can represent the class of convex problems.

We then define an oracle, denoted by $\mathcal{O}$, as a computational entity that can generate answers to our questions. An example is a gradient oracle, which can generate the value of the gradient of a function $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$, whenever we give it $\boldsymbol{x}$.

We denote by $\mathcal{M}$ the method, that is the algorithm that we are going to employ to solve our optimisation problem.

In distributed optimization, we will also need the communication network $\mathcal{G}$, either directed or undirected, which will affect how the algorithm behave.

Definition 2.1 (Performance of a distributed algorithm) *We call performance or* analytical complexity *of a method $\mathcal{M}$ for problem class $\mathcal{F}$ using the oracle $\mathcal{O}$ over a communication network $\mathcal{G}$, the number of calls to the oracle to find an approximate solution to $\mathcal{F}$ with an accuracy $\epsilon > 0$.*

The definition may be not very formal, but it will give us a clear direction for our theory. In particular we are interested in knowing how many iterations (i.e., how long do we have to wait) to obtain an approximate solution for a given optimization problem. This question is both extremely pragmatic and practical: you want to know if you have to wait one minute, or three days. But the question is also very theoretical: we want to find theoretical results that are valid for whole problem classes and communication networks.

Definition 2.2 (Computational complexity) *We call computational complexity of a method $\mathcal{M}$ for problem class $\mathcal{F}$ using the oracle $\mathcal{O}$ over a communication network $\mathcal{G}$, the maximal computational complexity of the oracle that the devices use.*

Definition 2.3 (Communication complexity) *We call communication complexity of a method $\mathcal{M}$ for problem class $\mathcal{F}$ using the oracle $\mathcal{O}$ over a communication network $\mathcal{G}$, the number of communication rounds that the method utilizes at each call of the oracle.*

We are now ready for our first two algorithms. We will specify them for the unconstrained problem,

$$\text{(Pu)} \qquad \min_{\boldsymbol{x}\in\mathbf{R}^n} \sum_{i=1}^{N} f_i(\boldsymbol{x}), \tag{2.2}$$

and then see how to extend them to the constrained case $\boldsymbol{x} \in X \subseteq \mathbf{R}^n$.

2.2 Decentralized gradient descent

2.2.1 The algorithm

We are ready to get back to our parallel update method. Define an undirected graph $\mathcal{G}$ and define scalar non-negative weights $w_{ij} \geqslant 0$ for each communication link. Endow each device with its own decision variable $\boldsymbol{x}^i \in \mathbf{R}^n$. Consider the following update:

Decentralized Gradient Descent (DGD)

- *Start with $\boldsymbol{x}_0^i \in \mathbf{R}^n$ at each device i and a stepsize sequence $\{\alpha_k\}_{k\in\mathbf{N}}$*
- *For all $k = 0, 1, \ldots$:*
    - **Communication step:** *send and receive $\boldsymbol{x}_k^i$ from your neighbors*
    - **Computation step:** *Each device i computes:*

$$\boldsymbol{x}_{k+1}^i = \sum_{j=1}^{N} w_{ij}\boldsymbol{x}_k^j - \alpha_k \nabla f_i(\boldsymbol{x}_k^i)$$

Here $w_{ij} = 0$ if device i and j are not connected.

DGD is our first distributed algorithm, that at each step asks every device to compute a modified gradient descent and the devices exchange that result with the neighbors. DGD does not need a global gathering node: it is peer-to-peer.

At every iteration, each device gets and sends $\boldsymbol{x}_k^j$ and $\boldsymbol{x}_k^i$ from and to the neighbors. Then, each device computes: $\boldsymbol{x}_{k+1}^i = \sum_{j=1}^N w_{ij}\boldsymbol{x}_k^j - \alpha_k \nabla f_i(\boldsymbol{x}_k^i), \quad \forall i$. If you notice that

$$\sum_{j=1}^{N} w_{ij}\boldsymbol{x}_k^j = w_{ii}\boldsymbol{x}_k^i + \sum_{j\in N_i} w_{ij}\boldsymbol{x}_k^j$$

then you see that each device needs only to communicate with the neighbors.

DGD is the first distributed algorithm that we see in this course and it does not need a server. Each device can compute its version of the optimal decision. Communication, embedded in the weights w_{ij} needs to happen synchronously and bi-directionally.

2.2.2 Matrix-vector representation and convergence

How do we interpret DGD? It is convenient to collect all the vectors and matrices into single entities. Let us look at the vector representation: define the iterates $\boldsymbol{y} = [\boldsymbol{x}^1; \boldsymbol{x}^2; \ldots; \boldsymbol{x}^N]$; define also the matrix $W = [w_{ij}]$ collecting all the weights and let $\mathbf{W} = W \otimes I_n$. To have everybody on the same page, recall that,

$$W = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1N} \\ w_{12} & w_{22} & \cdots & \vdots \\ \vdots & \vdots & w_{ij} & \vdots \\ \cdots & \cdots & \cdots & \cdots \end{bmatrix},$$

and the Kroenecker product implies:

$$\mathbf{W} = W \otimes I_n = \begin{bmatrix} w_{11} I_n & \cdots & w_{1N} I_n \\ \vdots & \ddots & \vdots \\ w_{1N} I_n & \cdots & w_{NN} I_n \end{bmatrix} \in \mathbf{R}^{Nn \times Nn}.$$

Finally, define the function,

$$F(\boldsymbol{y}) := \sum_{i=1}^{N} f_i(\boldsymbol{x}^i). \tag{2.3}$$

Notice that $F(\boldsymbol{y})$ is not the same as $f(\boldsymbol{x})$, since now the variables for each device are different.

Then, one iteration of DGD can be written as,

$$\boldsymbol{y}_{k+1} = \mathbf{W}\boldsymbol{y}_k - \alpha_k \nabla_{\boldsymbol{y}} F(\boldsymbol{y}_k). \tag{2.4}$$

Equation (2.4) is extremely compact (it hides all the complexity of implementing the method and fuses communication and computation step), but it is formally equivalent to the device-oriented version of DGD. The matrix-vector version (2.4) is useful for proofs and interpretation.

So, what DGD does? Effectively it is giving a copy of $\boldsymbol{x}$ to each device and then it mixes them via the mixing matrix $\mathbf{W}$. Is that very different from a gradient descent? We will see that it's not.

Consider a constant stepsize α, we can prove that the DGD iteration,

$$\boldsymbol{y}_{k+1} = \mathbf{W}\boldsymbol{y}_k - \alpha \nabla_{\boldsymbol{y}} F(\boldsymbol{y}_k) = \boldsymbol{y}_k - \alpha\Big[\nabla_{\boldsymbol{y}} F(\boldsymbol{y}_k) - \frac{1}{\alpha}(\mathbf{W} - I)\boldsymbol{y}_k\Big]$$

represents the standard gradient descent method on the problem,

$$\min_{\boldsymbol{y} \in \mathbf{R}^{Nn}} \alpha F(\boldsymbol{y}) + \frac{1}{2}\boldsymbol{y}^\top (I - \mathbf{W})\boldsymbol{y}.$$

This is quite a foundational result: we can interpret DGD as a gradient method on a modified problem. Let's look at this modified problem in terms of $\boldsymbol{x}$. We can write it as,

$$\min_{\boldsymbol{x}^1 \in \mathbf{R}^n, \ldots, \boldsymbol{x}^N \in \mathbf{R}^n} \sum_{i=1}^{N} f_i(\boldsymbol{x}^i) + \frac{1}{2\alpha} \sum_{i=1}^{N} \left(\|\boldsymbol{x}^i\|^2 - \sum_{j=1}^{N} w_{ij} \langle \boldsymbol{x}^i, \boldsymbol{x}^j \rangle \right)$$

where we can really see the original cost f changed so that each device has its own variable, and the penalization term that forces all the $\boldsymbol{x}^i$ to be as close as possible.

Assume now that,

Assumption 2.1 *The mixing matrix $W = [w_{ij}]$ is symmetric and doubly stochastic.*

Under Assumption 2.1, the eigenvalues of W are real and lie in the unit disk; they can be sorted in a non increasing order (as we have seen in Chapter 1.4.1 as,

$$1 = \lambda_1(W) \geqslant \lambda_2(W) \geqslant \ldots \geqslant \lambda_N(W) \geqslant -1.$$

Again, let the second largest magnitude of the eigenvalues of W be denoted as

$$\gamma := \max\{|\lambda_2(W)|, |\lambda_N(W)|\}.$$

We are now ready to characterize the properties of the algorithm. In particular, we will ask some basic questions: (1) When does $\boldsymbol{x}_k^i$ converge? (2) Does it converge to $\boldsymbol{x}^*$ (3) If not does consensus (i.e., $\boldsymbol{x}_k^i = \boldsymbol{x}_k^j$) hold asymptotically? (4) How do the properties of f_i and the network affect convergence?

Theorem 2.1 (DGD convergence) *Consider Problem (Pu) and its solution via a decentralized gradient descent algorithm with constant step size α, and doubly stochastic communication matrix W (Assumption 2.1) with $\gamma < 1$.*

Let convex functions f_i be L_i-smooth and let $L = \max_i\{L_i\}$. Let the mean value be

$$\bar{\boldsymbol{x}}_k = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{x}_k^i.$$

If α is chosen small enough and in particular $\leqslant O(1/L)$, then

1. *We obtain consensus with some errors in the sense that,*

$$\lim_{k\to\infty} \|\boldsymbol{x}_k^i - \bar{\boldsymbol{x}}_k\| = O(\frac{\alpha}{1-\gamma});$$

2. *We converge to a ball around the optimum $f^* = f(\boldsymbol{x}^*)$ as,*

$$\lim_{k\to\infty} f(\bar{\boldsymbol{x}}_k) - f^* = O(\frac{\alpha}{1-\gamma}).$$

Proof. *The proof is not difficult but long. For completeness is given in [YLY16].* ♣

What does the theorem mean? First of all, we observe two things: convergence of the mean value and consensus around the mean, this is a general characteristic of distributed optimization. We also see a dependency on the graph entering via its spectrum γ, and in particular (here) the second largest eigenvalue (in modulus), and its spectral gap $1/(1-\gamma)$.

Second, we see the presence of errors, which is not so surprising, since we are changing the cost function by adding a penalty term.

The theorem can be specified to the case in which the functions f_i's are strongly convex (which is easier to handle and one obtains linear convergence to an error ball), but it is much harder to work with non-doubly stochastic matrices (i.e., when one models communication issues). The reader is referred to [YLY16] for a complete account.

And finally, when the stepsize α is diminishing, we can obtain a zero error bound, but the analysis is more complicated, and you require typically extra assumptions, see for instance [NO09].

2.2.3 Network effects

It is interesting now to look at how the error term depends on the communication network, for an example of weight matrix. It turns out, we already did all the required job in Theorem 1.5 for the Lazy Metropolis weights! For instance, for a full graph with weights $\tau = 2$, because you can choose,

$$W = \frac{I}{2} + \frac{1}{2}\begin{bmatrix} \frac{1}{N} & \cdots & \frac{1}{N} \\ \vdots & \ddots & \vdots \\ \frac{1}{N} & \cdots & \frac{1}{N} \end{bmatrix},$$

which has one eigenvalue at 1 and all the rests at 1/2. So the error does not depend on the number of devices, it is just a constant factor.

In the worst case, for a line graph, $\tau = O(N^2)$, and then the error depends on the number of devices in a quadratic fashion. The more the devices, the network effect is squared. So the error can get pretty bad for loosely connected graphs running DGD.

2.2.4 Constraints

Once we have understood the link between DGD and the modified optimization problem,

$$\min_{\boldsymbol{y}\in\mathbf{R}^{Nn}} \alpha F(\boldsymbol{y}) + \frac{1}{2}\boldsymbol{y}^\top(I-\mathbf{W})\boldsymbol{y}, \tag{2.5}$$

then adjusting it to account for constraints is rather straightforward. Let us assume that each device has to verify a local constraint set X_i, and let the global constraint being the cartesian product of such constraints. In this context, we write $\boldsymbol{x}^i \in X_i \subseteq \mathbf{R}^n$, and we set

$$Y = X_1 \times X_2 \times \cdots \times X_n.$$

This global set is totally decoupled. Now consider the problem,

$$\min_{\boldsymbol{y}\in Y\subseteq\mathbf{R}^{Nn}} \alpha F(\boldsymbol{y}) + \frac{1}{2}\boldsymbol{y}^\top(I-\mathbf{W})\boldsymbol{y}.$$

This aims at solving the constrained problem,

$$(\mathsf{Pc}) \quad \underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}} \sum_{i=1}^N f_i(\boldsymbol{x}), \text{ subject to } \boldsymbol{x}\in X_i, \forall i. \tag{2.6}$$

Problem (Pc) is an embodiment of the master problem (2.1), one that we can solve with a projected version of DGD.

Projected Decentralized Gradient Descent (P-DGD)

- *Start with* $\boldsymbol{x}_0^i \in \mathbf{R}^n$ *at each device* i *and a stepsize sequence* $\{\alpha_k\}_{k\in\mathbf{N}}$
- *For all* $k = 0, 1, \dots$ *:*
  - **Communication step:** *send and receive* $\boldsymbol{x}_k^i$ *from your neighbors*
  - **Computation step:** *Each device* i *computes:*

$$\boldsymbol{x}_{k+1}^i = \mathsf{P}_{X_i}\left[\sum_{j=1}^N w_{ij}\boldsymbol{x}_k^j - \alpha_k \nabla f_i(\boldsymbol{x}_k^i)\right]$$

As one can see, the algorithm is still peer-to-peer, since the local constraints stay local. Convergence of the method is very similar to the classical DGD and omitted here. The interesting thing to remember is to get back to the general formulation (2.5) and reason about how to add new features. We will see that in a few examples shortly.

2.2.5 Other variants over the classical DGD

As you may wonder, DGD is not the only first-order gradient descent method, even though the basic error estimates will stay the same (i.e., the asymptotic non-zero error for non-vanishing step size will stay the same).

We cite here another example (which will encounter later) which employs the recursion:

$$\boldsymbol{y}_{k+1} = \mathbf{W}(\boldsymbol{y}_k - \alpha\nabla_{\boldsymbol{y}} F(\boldsymbol{y}_k)).$$

Let's call it **distributed gradient descent** for us, since it has many different names. This method is also peer-to-peer and it solves the problem,

$$\min_{\boldsymbol{y}\in\mathbf{R}^{Nn}} \frac{1}{2}\boldsymbol{y}^\top(I-\mathbf{W})\boldsymbol{y} + \alpha\mathbf{W}F(\boldsymbol{y}).$$

The convergence proof is more involved, but distributed gradient descent has the property that iff $\mathbf{W}$ is fully connected and $W_{ij} = 1/N$ for all i, j, then $\boldsymbol{y}_{k+1} \to \boldsymbol{y}^*$.

Proof. *Since* $\mathbf{W}$ *averages all the elements, all the* $\boldsymbol{x}_{k+1}^i$ *are the same, and* $\alpha\nabla_{\boldsymbol{y}}F(\boldsymbol{y}_{k+1}) = (\alpha/N)\sum_i \nabla_{\boldsymbol{x}} f_i(\boldsymbol{x}_{k+1})$*, that is the standard gradient descent.* ♣

This is the only real property that is different for distributed gradient descent with DGD.

2.2.6 Further examples

Example 2.1 *In the protocol of giving each device a copy if $\boldsymbol{x}$ and then force each $\boldsymbol{x}^i$ to be the same, we could consider trying to solve the problem,*

$$\min_{\boldsymbol{x}^i \in \mathbf{R}^n, i=1,\dots,N} \alpha \underbrace{\boxed{\sum_{i=1}^N f_i(\boldsymbol{x}^i)}}_{(a)} + \underbrace{\boxed{\frac{1}{2(\max_i(d^i)+1)} \sum_{i\sim j} \|\boldsymbol{x}^i - \boldsymbol{x}^j\|^2}}_{(b)} \equiv$$

$$\min_{\boldsymbol{y}\in\mathbf{R}^{Nn}} \alpha F(\boldsymbol{y}) + \frac{1}{2(\max_i(d^i)+1)} \boldsymbol{y}^\top (\mathcal{L}_\mathcal{G} \otimes I_n)\boldsymbol{y},$$

where we recall that $\mathcal{L}_\mathcal{G}$ is the Laplacian of the graph and d^i is the degree of device i. The term (a) is the original problem decoupled over the devices. The term (b) is a forcing term, pushing all the $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$ to be the same, if i and j share an edge (i.e., $i \sim j$). It is now easy to see that the above is an example of DGD, for the choice,

$$\mathbf{W} = I - \frac{1}{\max_i(d^i)+1} \mathcal{L}_\mathcal{G} \otimes I_n,$$

which is doubly stochastic for Example 1.6.

Example 2.2 (Exam 2024) Overlapping images

In a modern medical imaging techniques, several sensors take partial and overlapping images of the same body element. Since the amount of data that each sensor receives is big, there is much research around how one could find the overall image by distributed optimization methods.

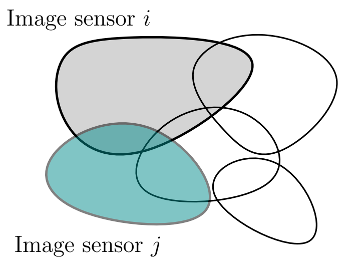


Figure 2.1. *Partial images and their overlaps.*

Consider Figure 2.1, where we have indicated the partial images of sensors i and j, and their overlap. Consider the measurements for sensor i as $y_i \in \mathbf{R}^{m_i}$ and the decision variables $x_i \in \mathbf{R}^{n_i}$. The problem can be modeled as solving the optimization problem involving N sensors,

$$\min_{x_i \in \mathbf{R}^{n_i}, i=1,\dots,N} \quad \frac{1}{2}\sum_{i=1}^N \|y_i - C_i x_i\|_2^2,$$
$$\text{subject to: } V^{ij}x_i = V^{ji}x_j, \quad \forall i \sim j,$$

where $C_i \in \mathbf{R}^{m_i \times n_i}$ is a matrix ($m_i \leqslant n_i$), and matrices $V^{ij} \in \mathbf{R}^{\ell_{ij}\times n_i}$, $V^{ji} \in \mathbf{R}^{\ell_{ij}\times n_j}$ are selection matrices that describe the overlap between decision x_i and x_j. Finally $i \sim j$ tells which sensor has overlaps with which other one. We assume that if two sensors have overlaps, then they can message each other.

1. *Describe the problem: is it convex? Is the cost smooth? Strongly convex?*
2. *Start by removing the constraints and adding to the cost the penalty $\sum_{i\sim j} \frac{1}{2\alpha}\|V^{ij}x_i - V^{ji}x_j\|^2$ and derive a first-order decentralized gradient descent scheme.*
3. *Describe the decentralized method that you have obtained in the previous point. In particular, how many communication rounds per iteration you need? What do you exchange? Can you derive a convergence result, and if so, is it to the true optimizer or to an approximate one?*

Sketched solutions.

1. The problem is convex and smooth. Since $m_i \leqslant n_i$, the problem is not generally strongly convex.

2. We follow what it is asked and write the cost as,

$$F(\boldsymbol{x}) := \frac{1}{2}\sum_{i=1}^{N} \|y_i - C_i x_i\|_2^2 + \sum_{i\sim j} \frac{1}{2\alpha}\|V^{ij}x_i - V^{ji}x_j\|^2.$$

   I can now write a first-order gradient scheme as,

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha\nabla_{\boldsymbol{x}} F(\boldsymbol{x}_k).$$

   In particular, we can update each x_i as,

$$x_{i,k+1} = x_{i,k} - \alpha\left(C_i^\top[C_i x_{i,k} - y_i] + \sum_{j\in\mathcal{N}_i}\frac{1}{\alpha}V^{ij,\top}[V^{ij}x_{i,k} - V^{ji}x_{j,k}]\right).$$

3. The decentralized method would need to be initialized so that each device i has all the V^{ij} and V^{ji} is needs. Then at each round each device shares $x_{i,k}$ to its neighbors and compute the update $x_{i,k+1}$.

   I need one communication round per iteration, exchanging $x_{i,k}$. The above scheme is a standard gradient scheme on smooth convex function. So picking $\alpha < 2/L$, we can reach a $O(1/k)$ convergence guarantee in terms of function values. We can also accelerate the scheme with Nesterov's acceleration, obtaining $O(1/k^2)$. Here we converge to an optimizer of the modified problem, so it is an approximate optimizer with approximation due to α.

Example 2.3 (Exam 2023) *Consider the problem,*

$$(P') \qquad \min_{x_1,x_2\in\mathbf{R}} \sum_{i=1}^{2} f_i(x_i) + \epsilon|x_1 - x_2|,$$

with $\epsilon > 0$ and smooth, strongly convex f_i.

1. *Apply a (sub)-gradient descent to it using a constant stepsize α, and remember that $\partial_{x_1}|x_1 - x_2| = sign(x_1 - x_2)$. Show that the resulting algorithm is distributed (in the sense that f_i is not shared).*

2. *What is the convergence and convergence rate you can expect for the algorithm developed at the previous point?*

3. *Show that, if you assume that $\|\nabla f_i(x)\| \leqslant B$ for all $x \in \mathbf{R}$, and you choose $\epsilon > B$, then at convergence, $x_1^* = x_2^*$, and therefore the penalization is exact. This means that you have no error.*

   (Hint) Use the optimality conditions.

4. **[Bonus question]** *A sub-gradient algorithm is very slow. Write a proximal gradient algorithm and show that is distributed and converges linearly.*

Solution.

1. I follow what is asked, I compute the subgradient scheme,

$$x_{i,k+1} = x_{i,k} - \alpha g_{i,k}, \qquad g_{i,k} \in \nabla_{x_i} f_i(x_{i,k}) + \epsilon\text{sign}(x_{1,k} - x_{2,k}).$$

   This leads to a distributed algorithm, since each device shares $x_{i,k}$ but not f_i.

2. It's a subgradient scheme, so at best I converge as $O(1/\sqrt{k})$ in terms of function values.

3. I use the hint. Optimality implies,

$$\nabla f_1(x_1^*) + \epsilon\text{sign}(x_1^* - x_2^*) \ni 0, \qquad \nabla f_2(x_2^*) + \epsilon\text{sign}(x_1^* - x_2^*) \ni 0.$$

   Since $\text{sign}(x_1^* - x_2^*)$ is bounded in $[-1, 1]$, assuming $\|\nabla f_i(x)\| \leqslant B < \epsilon$, then if $x_1^* \neq x_2^*$, then the equation above could not be satisfied. Then at optimality, it has to be $x_1^* = x_2^*$.

4. I can write a proximal gradient as,

$$x_{i,k+1} = \mathrm{Prox}_{\epsilon|x_1 - x_2|}\{x_{i,k} - \alpha \nabla f_i(x_{i,k})\},$$

   and derive a closed form solution for the proximal operator. The linear convergence follows from 4OPT2.

2.3 Gradient tracking

In the previous section, we have seen that DGD has a fundamental trade-off between accuracy and performance. A constant stepsize selection will lead to a fast convergence but to an error term, and a vanishing stepsize will recover zero error, but with a slower rate. We have argued that this is a general feature of distributed first-order algorithms in the primal space. This section will teach us how to do better and obtain linear convergence to zero error.

Consider the following distributed algorithm.

Decentralized Gradient Tracking

- *Initialize* $\boldsymbol{x}_0^i \in \mathbf{R}^n$, $\boldsymbol{g}_0^i \in \mathbf{R}^n$ *at each device and a stepsize* α
- *For all* $k = 0, 1, \ldots$
  - **Communication step:** *send and receive* $\boldsymbol{x}_k^i$ *and* $\boldsymbol{g}_k^i$ *from your neighbors*
  - **Computation step:** *each device* i *computes:*

$$\begin{aligned} \boldsymbol{x}_{k+1}^i &= \sum_{j=1}^N w_{ij} \boldsymbol{x}_k^j - \alpha \boldsymbol{g}_k^i, \\ \boldsymbol{g}_{k+1}^i &= \sum_{j=1}^N w_{ij} \boldsymbol{g}_k^j + (\nabla f_i(\boldsymbol{x}_{k+1}^i) - \nabla f_i(\boldsymbol{x}_k^i)). \end{aligned}$$

Decentralized Gradient Tracking, or Gradient Tracking for short, is a peer-to-peer algorithm that requires twice the amount of communication and twice the computations of DGD. It maintains a supporting vector $\boldsymbol{g}^i$ which estimate the true gradient. For the readers that are accustomed to the theory of control, the term, $\sum_{j=1}^N w_{ij} \boldsymbol{g}_k^j + (\nabla f_i(\boldsymbol{x}_{k+1}^i) - \nabla f_i(\boldsymbol{x}_k^i))$ can be interpreted as an "integrator" which helps achieving zero tracking error.

2.3.1 Matrix-vector representation and convergence

Once again, we can rewrite the gradient tracking scheme in its matrix-vector representation. Denote with $\boldsymbol{y} = [\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N]$ and $\boldsymbol{q} = [\boldsymbol{g}^1, \ldots, \boldsymbol{g}^N]$. Then, the gradient tracking method reads,

$$\begin{cases} \boldsymbol{y}_{k+1} = \mathbf{W} \boldsymbol{y}_k - \alpha \boldsymbol{q}_k, \\ \boldsymbol{q}_{k+1} = \mathbf{W} \boldsymbol{q}_k + \nabla_{\boldsymbol{y}} F(\boldsymbol{y}_{k+1}) - \nabla_{\boldsymbol{y}} F(\boldsymbol{y}_k) \end{cases} \tag{2.7}$$

To prove convergence of the method, the representation (2.7) is most useful.

Let us start by defining the mean quantity, $\bar{\boldsymbol{x}}_k = \frac{1}{N} \sum_{i=1}^N \boldsymbol{x}_k^i$. Then, the following theorem is in place.

Theorem 2.2 (Gradient Tracking convergence) *Consider Problem (Pu) and its solution via a gradient tracking algorithm with constant step size* α, *and doubly stochastic communication matrix* W *with* $w_{ij} \geqslant 0$ *and* $\gamma < 1$. *Let convex functions* f_i *be* L*-smooth and* m*-strongly convex. If* α *is chosen small enough and in particular* $\leqslant O(1/L)$, *then*

1. *Each device reaches a consensus:*

$$\lim_{k \to \infty} \|\boldsymbol{x}_k^i - \bar{\boldsymbol{x}}_k\| = 0;$$

2. The mean convergences to the true optimizer:

$$\lim_{k\to\infty} \|\bar{\boldsymbol{x}}_k - \boldsymbol{x}^*\| = 0.$$

And convergence is linear.

Proof. *The full proof can be found in [NNC19]. We give here a brief account for $n = 1$: it is based on writing the error as the vector:*

$$\boldsymbol{v}_k := \begin{bmatrix} \|\boldsymbol{y}_k - \bar{\boldsymbol{x}}_k \mathbf{1}\| \\ \|\boldsymbol{q}_k - \bar{\boldsymbol{g}}_k \mathbf{1}\| \\ \|\bar{\boldsymbol{x}}_k - \boldsymbol{x}^*\| \end{bmatrix}.$$

where $\bar{\boldsymbol{x}}_k$ and $\bar{\boldsymbol{g}}_k$ are the average values of $\boldsymbol{x}_k^i$ and $\boldsymbol{g}_k^i$, respectively. If $\boldsymbol{v}_k$ goes to zero, then we obtain convergence as desired.

Then, combining smoothness and strong convexity property, the authors in [NNC19] prove that,

$$\boldsymbol{v}_{k+1} \leqslant J(\alpha)\boldsymbol{v}_k, \qquad J(\alpha) := \begin{bmatrix} \gamma & \alpha & 0 \\ \left(L\|W - I\| + \alpha L^2\sqrt{N}\right) & \gamma + \alpha L & \alpha L^2\sqrt{N} \\ \gamma\frac{L}{\sqrt{N}} & 0 & \theta \end{bmatrix}. \tag{2.8}$$

Recall that $\gamma = \max|\lambda_2(W)|, |\lambda_N(W)|$ and we have called $\theta = \max(|1 - \alpha m/N|, |1 - \alpha L/N|)$. Since $m \leqslant L$ and $\alpha \leqslant N/L$, it follows that $\theta = 1 - \alpha m/N$. Thus, we can express $J(\alpha)$ as the sum of two structured matrices as follows

$$J(\alpha) = \begin{bmatrix} \gamma & 0 & 0 \\ L\|W - I\| & \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} + \alpha \begin{bmatrix} 0 & 1 & 0 \\ L^2\sqrt{N} & 1 & L^2\sqrt{N} \\ \frac{L}{\sqrt{N}} & 0 & -m/N \end{bmatrix}.$$

Being $\gamma < 1$ and due to the triangular structure of the left matrix, we can conclude that it has spectral radius equal to 1. We use now [HJ12, Theorem 6.3.12] for a small perturbation $\alpha > 0$. For $\alpha = 0$, the eigenvalues of $J(\alpha)$ are 1 and $\gamma < 1$. By continuity of the eigenvalues w.r.t. the matrix coefficients, for small enough α, the eigenvalues < 1 will remain < 1. For the single eigenvalue 1 with left and right eigenvector col$(0, 0, 1)$ one can use [HJ12, Theorem 6.3.12(a)], to say that the corresponding eigenvalue of $J(\alpha)$, say $\lambda(\alpha)$, will be

$$|\lambda(\alpha) - 1 + \alpha m/N| \leqslant \alpha\varepsilon$$

for any $\varepsilon > 0$ and sufficiently small α. If then one selects e.g., $\varepsilon = \frac{m}{2N}$, then $\lambda(\alpha) \in [1 - \frac{3\alpha m}{N}, 1 - \frac{\alpha m}{N}]$, meaning that there exists a small enough α, for which all the eigenvalues of $J(\alpha)$ are all strictly less than one, and therefore the spectral radius of $J(\alpha)$, say ρ, becomes strictly less than one.

Hence, we have $\boldsymbol{v}_{k+1} \leqslant \rho\,\boldsymbol{v}_k$ with $\rho \in (0, 1)$. Thus, $\|\boldsymbol{v}_k\| \to 0$ as $k \to \infty$ with linear rate, and the proof follows. ♣

The proof of the theorem is interesting. For well-conditioned functions and poorly connected graphs, one can also argue that the spectral radius ρ of $J(\alpha)$ will be dominated by the term γ, and for small enough α, we can also write,

$$\boldsymbol{v}_k \lesssim \gamma^k \boldsymbol{v}_0.$$

This implies that to reach a certain accuracy ϵ, one needs,

$$k \geqslant O\left(\frac{1}{1-\gamma}\log(1/\epsilon)\right),$$

iterations. One should not be surprised of the appearance of the spectral gap $\frac{1}{1-\gamma}$.

Gradient Tracking achieves what DGD could not, linear convergence to zero-error. This comes with some limitations however. First, it costs twice in communication and computation. Second, generalizations to non-strongly convex functions, other communication patters (i.e., non doubly-stochastic), and more importantly to constrained problems, are much harder to obtain with Gradient Tracking and they are very involved, see [NNC19] for an account.

We also communicate gradient information in terms of $\boldsymbol{g}^i$, this is in general bad for privacy (as we will see later). In summary, we have a peer-to-peer primal method that converges exactly, but the price to pay may be quite high in some applications and researchers still use DGD very often.

2.4 The EXTRA algorithm *

Research paper 3 *Wei Shi, Qing Ling, Gang Wu, and Wotao Yin,* EXTRA: An Exact First-Order Algorithm for Decentralized Consensus Optimization*, SIAM Journal on Optimization, Vol. 25 (2), 2015*

2.5 The DIGing algorithm *

Research paper 4 *Angelia Nedić, Alex Olshevsky, and Wei Shi,* Achieving Geometric Convergence for Distributed Optimization Over Time-Varying Graphs*, SIAM Journal on Optimization, Vol. 27 (4), 2017*

2.6 Distributed Newton's method(s) *

Research paper 5 *Mokhtari, Aryan, Qing Ling, and Alejandro Ribeiro.* Network Newton distributed optimization methods. *IEEE Transactions on Signal Processing, Vol. 65 (1), 2016.*

2.7 Numerical examples

2.7.1 Distributed Kernel ridge regression

We try now DGD and Gradient Tracking on the Kernel example of Chapter 1.5.1. In particular, we consider $N = 5$ devices, and we set $\nu = 1.0$. Figure 2.2 captures the convergence of the methods with the Metropolis weights on the depicted undirected graph. In the figure, we also indicate the stepsize (here as s, since α is the decision variable). The theory is confirmed, with the non-zero error convergence of DGD and the zero convergence of Gradient Tracking.

In Figure 2.3, we report the reconstruction of the function, by device 1 using Gradient Tracking, with respect to the centralized reconstruction (in orange).

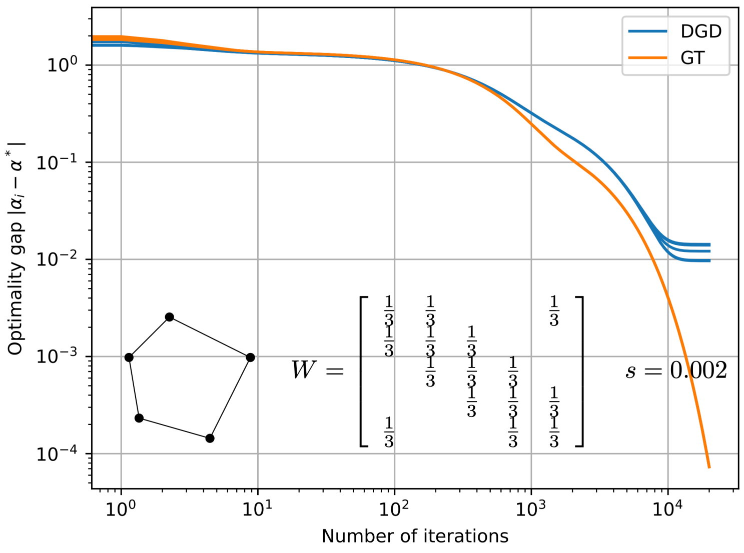


Figure 2.2. *Convergence in terms of $\|\alpha^i - \alpha^*\|$ for DGD and Gradient Tracking on the Kernel ridge regression example.*

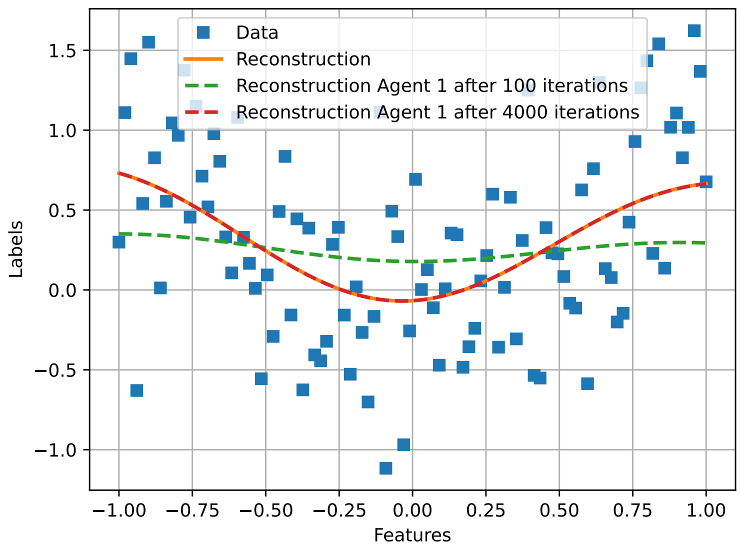


Figure 2.3. *Decentralized reconstuction by using Gradient Tracking on the Kernel ridge regression example.*

Chapter 3

Consensus and Network dependencies

3.1 The special case of average consensus

This chapter is devoted to the study of the special case of quadratic functions. This will help us understand the concept of *reaching consensus* among devices, and also study time-varying graphs and graph dependencies in a easier and safer setting.

As such, for most of the chapter, we will look at problems of the form,

$$\underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}}\, \frac{1}{2}\sum_{i=1}^{N}\|\boldsymbol{x}-\boldsymbol{u}_i\|_2^2, \tag{3.1}$$

where $\boldsymbol{u}_i \in \mathbf{R}^n$ are data proper to device i. Since the optimizer of such simple problem is the average of the data points, i.e., $\boldsymbol{x}^* = \frac{1}{N}\sum_{i=1}^{N}\boldsymbol{u}_i$, then the question is how to ensure that the devices can reach a consensus on what the average is.

If we were to employ DGD on this problem, we already know that we would not be able to reach consensus with a constant stepsize. It is easy to see that, since the DGD iteration is,

$$\boldsymbol{x}_{k+1}^i = \sum_{j=1}^{N} w_{ij}\boldsymbol{x}_k^j - \alpha(\boldsymbol{x}_k^i - \boldsymbol{u}_i),$$

and if all the $\boldsymbol{x}_k^i$ were the same (and equal to $\boldsymbol{x}^*$), the iterate $k+1$ would bring us away from it by the term $(1-\alpha)\boldsymbol{x}^* + \alpha\boldsymbol{u}_i$. So, instead of DGD, we could employ only a consensus iteration as follows.

Average (linear) consensus

- *Start by initializing* $\boldsymbol{x}_0^i = \boldsymbol{u}_i$
- *Iterate for all* $k = 0, 1, \dots$
    - **Communication step:** *send and receive* $\boldsymbol{x}_k^i$ *to and from your neighbors*
    - **Computation step:** *each device computes,*

$$\boldsymbol{x}_{k+1}^i = \sum_{j=1}^{N} w_{ij}\boldsymbol{x}_k^j.$$

The algorithm can be put into the matrix-vector form, as usual, by introducing the stacked vector $\boldsymbol{y}$ and the matrix $\mathbf{W}$ as,

$$\boldsymbol{y}_{k+1} = \mathbf{W}\boldsymbol{y}_k, \qquad \boldsymbol{y}_0 = [\boldsymbol{u}_1^\top, \dots, \boldsymbol{u}_N^\top]^\top. \tag{3.2}$$

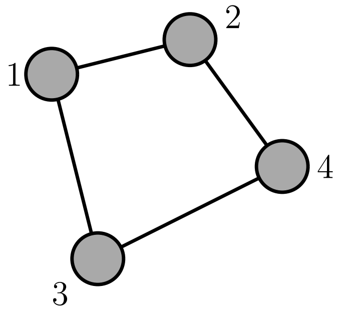


Figure 3.1. *A connected undirected graph of four devices.*

In this chapter, we study the more general recursion,

$$\boldsymbol{y}_{k+1} = \mathbf{W}_k \boldsymbol{y}_k, \qquad \boldsymbol{y}_0 = [\boldsymbol{u}_1^\top, \dots, \boldsymbol{u}_N^\top]^\top, \tag{3.3}$$

where we allow the communication weights to change with the iteration counter to simulate time-varying communication graphs.

This chapter follows mainly from [XBL06, NOR18].

3.2 Reaching consensus

So, let's look at the recursion (3.3) for arbitrary $\mathbf{W}_k$. The aim is to understand a bit better the results we found in distributed optimization for doubly-stochastic matrices, as well as why considering asynchronous and directed communication is problematic. This also makes us understand why purely gossip communication protocols (directed, asynchronous) are hard to work with. To keep notation and proofs easier to understand, we consider the scalar case, N devices $\boldsymbol{x}^i \in \mathbf{R}, \boldsymbol{y} \in \mathbf{R}^N$ as collection of all the scalar $\boldsymbol{x}^i$. This goes without loss of generality, since the vector case can be derive analogously with the use of Kroenecker product formulas.

Note that the theory we present here is related to stochastic matrix theory, and Markov chains.

3.2.1 Doubly-stochastic, time-invariant case

Let us analyze the easiest case of doubly-stochastic and time-invariant $\mathbf{W}$. We have already seen in DGD and Gradient tracking that this is a favourable setting. Figure 3.1 depicts a sample 4 device graphs capturing the setting.

We can prove the following properties for the recursion (3.3):

1. The average is preserved

$$\frac{\mathbf{1}}{N}^\top \boldsymbol{y}_{k+1} = \frac{\mathbf{1}}{N}^\top \mathbf{W}\boldsymbol{y}_k = \frac{\mathbf{1}}{N}^\top \boldsymbol{y}_k = \dots = \left(\sum_{i=1}^N \boldsymbol{u}_i\right)\frac{\mathbf{1}}{N},$$

   that is, each iteration preserve the average of the initial condition (which is the data).

2. Consensus is preserved. If all $\boldsymbol{x}_k^i = \bar{\boldsymbol{x}}$ are the same then,

$$\boldsymbol{y}_{k+1} = \mathbf{W}\mathbf{1}\bar{\boldsymbol{x}} = \mathbf{1}\bar{\boldsymbol{x}},$$

   which implies that if we reach consensus, we stay at consensus.

With this in place, the following theorem can be proven.

Theorem 3.1 *Consider the recursion,* $\boldsymbol{y}_{k+1} = \mathbf{W}\boldsymbol{y}_k$*. Assume that* $\mathbf{W}$ *is doubly stochastic, the communication graph is connected, and* $\|\mathbf{W} - \mathbf{1}\mathbf{1}^\top/N\| < 1$*. Let* $\lambda_i(\mathbf{W})$ *be the eigenvalues of* $\mathbf{W}$ *sorted in descending order, with*

$$1 = \lambda_1(\mathbf{W}) \geqslant \lambda_2(\mathbf{W}) \geqslant \lambda_N(\mathbf{W}) = -1,$$

and set $\gamma := \max\{|\lambda_2(\mathbf{W})|, |\lambda_N(\mathbf{W})|\}$. *Then linearly*

$$\lim_{k\to\infty} \boldsymbol{y}_k = \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{y}_0,$$

and therefore $\boldsymbol{x}_k^i \to \boldsymbol{x}^* = \frac{1}{N}\sum_{i=1}^N \boldsymbol{u}_i$. *Furthermore the convergence rate is* γ.

Proof. *Define the error* e_k *at time* k *as* $e_k = \boldsymbol{y}_k - \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{y}_0$. *For our recursion we have,*

$$e_{k+1} = \boldsymbol{y}_{k+1} - \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{y}_0 = W\boldsymbol{y}_k - \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{y}_0 = (\mathbf{W}^{k+1} - \frac{\mathbf{1}\mathbf{1}^\top}{N})e_0.$$

We now use the double stochasticity of $\mathbf{W}$ *to prove,*

$$\mathbf{W}^2 - \frac{\mathbf{1}\mathbf{1}^\top}{N} = \mathbf{W}^2 - 2\frac{\mathbf{1}\mathbf{1}^\top}{N} + \frac{\mathbf{1}\mathbf{1}^\top}{N} = (\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N})^2,$$

and similarly, $\mathbf{W}^{k+1} - \frac{\mathbf{1}\mathbf{1}^\top}{N} = (\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N})^{k+1}$.

Connectedness of the communication graph ensures we can choose a $\mathbf{W}$ *such that the spectral radius* $\|\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N}\| < 1$. *With this in place,*

$$\lim_{k\to\infty}\|e_k\| \leqslant \lim_{k\to\infty}\left\|\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N}\right\|^k \|e_0\| = 0.$$

To prove the rate, consider the matrix, $\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N}$. *Its eigenvalues are*

$$\mathit{eig}(\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N}) = \begin{cases} \mathit{eig}(W) & \textit{for all eigenvect.} \neq \mathbf{1} \\ 0 & \textit{for all eigenvect.} = \mathbf{1} \end{cases},$$

which, for us, means that the dominant eigenvalue of $\mathbf{W} - \frac{\mathbf{1}\mathbf{1}^\top}{N}$ *is* γ, *and thereby the rate, i.e.,* $e_{k+1} = O(\gamma^{k+1})$. ♣

Theorem 3.1 tells you that if $\mathbf{W}$ is doubly stochastic, then, eventually (by communicating and communicating), you approximate the average matrix $\frac{\mathbf{1}\mathbf{1}^\top}{N}$, exactly. This is good news. However, in general, as we saw in DGD, the interleaved scheme: consensus plus optimization, yield a non-zero error, so the harder setting is harder indeed.

Also, it shouldn't come as a surprise that typically the error in distributed optimization, where we do a mixing plus a term is $\sim O(\frac{1}{1-\gamma})$, where $1-\gamma$ is as always the spectral gap. Recall that $\gamma \geqslant 1$ if the graph is not connected. In fact, consider the relationship: $e_k = O(\gamma^k)$. To reach a certain accuracy ϵ, then the number of iterations needed can be upper bounded as,

$$k \geqslant O\left(\frac{1}{1-\gamma}\log(1/\epsilon)\right).$$

Proof. *The number of iterations follows from* $O(\gamma^{k+1}) \leqslant \epsilon$, *which implies, for a positive constant* β,

$$\gamma^k \leqslant \beta\epsilon \iff \exp(-k\log(1/\gamma)) \leqslant \beta\epsilon.$$

Use now the fact that $\log(1/\gamma) \geqslant 1-\gamma$, *for all* γ, *both of which are non-negative for* $\gamma \leqslant 1$. *With this,*

$$\exp(-k(1-\gamma)) \leqslant \beta\epsilon \implies \exp(-k\log(1/\gamma)) \leqslant \beta\epsilon,$$

or $k \geqslant O(\frac{1}{1-\gamma}\log(1/\epsilon))$. ♣

As explained in Chapter 1.4.1, for given doubly stochastic matrices, we can characterize the spectral gap, and therefore estimate the number of iteration needed. For the Lazy Metropolis weights, the job is done in Theorem 1.5, and in particular,

1. Line graphs: $k \geqslant O(N^2\log(1/\epsilon))$;

2. Full graphs: k; DO
3. Generic graphs: $\tau = O(N^2 \log(1/\epsilon))$;
4. Erdös-Rényi random graphs (with high probability): $k \geqslant O(\log(1/\epsilon))$.

See also [NOR18].

3.2.2 Laplacian weights

It is interesting to look back at Example 1.6 on the Laplacian weights, setting $W = I - \varepsilon \mathcal{L}_{\mathcal{G}}$ for $\varepsilon < \max_i\{d^i\}$.

Let us look first at the dynamical system,

$$\frac{\mathrm{d}\boldsymbol{x}(t)}{\mathrm{d}t} = -\mathcal{L}_{\mathcal{G}}\boldsymbol{x}(t), \tag{3.4}$$

for state $\boldsymbol{x}(t) \in \mathcal{L}_2(\mathbf{R}^N)$ and initial condition $\boldsymbol{x}(0)$. Then the system is asymptotically stable iff all the eigenvalues of the system matrix $-\mathcal{L}_{\mathcal{G}}$ are in the negative half-plane.

The Laplacian has eigenvalues that can be ordered as,

$$0 = \lambda_1(\mathcal{L}_{\mathcal{G}}) \leqslant \lambda_2(\mathcal{L}_{\mathcal{G}}) \leqslant \cdots \leqslant 2\max_i\{d^i\},$$

for Gershgorin circle theorem (Cf. Theorem 1.4). The second smallest eigenvalue $\lambda_2(\mathcal{L}_{\mathcal{G}})$ is strictly positive iff the graph is connected, and it is named as algebraic connectivity.

As such, the system (3.4) is only marginally stable. For any initial condition it converges to the average of the initial conditions.

Theorem 3.2 *Consider system (3.4) for the initial condition $\boldsymbol{x}(0)$ and connected graph $\mathcal{G}$. Then,*

$$\lim_{t\to\infty} \boldsymbol{x}(t) = \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{x}_i(0),$$

where $\boldsymbol{x}_i(0)$ is the i-th component of $\boldsymbol{x}(0)$.

Proof. *The initial condition can be decomposed as $\boldsymbol{x}(0) = \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{x}(0) + (I - \frac{\mathbf{1}\mathbf{1}^\top}{N})\boldsymbol{x}(0)$. Then,*

$$\boldsymbol{x}(t) = e^{-\mathcal{L}_{\mathcal{G}}\,t}\left(\frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{x}(0) + (I - \frac{\mathbf{1}\mathbf{1}^\top}{N})\boldsymbol{x}(0)\right) = \frac{\mathbf{1}\mathbf{1}^\top}{N}\boldsymbol{x}(0) + \underbrace{e^{-\mathcal{L}_{\mathcal{G}}\,t}(I - \frac{\mathbf{1}\mathbf{1}^\top}{N})\boldsymbol{x}(0)}_{\to 0 as\ t\to\infty},$$

where we have used the fact that the Laplacian has one eigenvalue at 0 with eigenvector $\mathbf{1}$. The rate at which the system converges is upper bounded by the algebraic connectivity. ♣

With this in place, the discrete-time recursion,

$$\boldsymbol{y}_{k+1} = (I - \varepsilon\mathcal{L}_{\mathcal{G}})\boldsymbol{y}_k,$$

is a forward Euler representation of the dynamical system (3.4), which also converges to the mean of the initial values $\boldsymbol{y}_0$, provided that ε is chosen sufficiently small.

3.2.3 Doubly-stochastic, time-varying case

Let us move now to the case in which the matrices $\mathbf{W}_k$ stay doubly stochastic but they vary in time. This is the case which is depicted in Figure 3.2, where at each time k edges are randomly selected and the devices whose edges are active communicate with each other. This setting is also called edge asynchronous. As you see in Figure 3.2, the instantaneous graphs may be not connected.

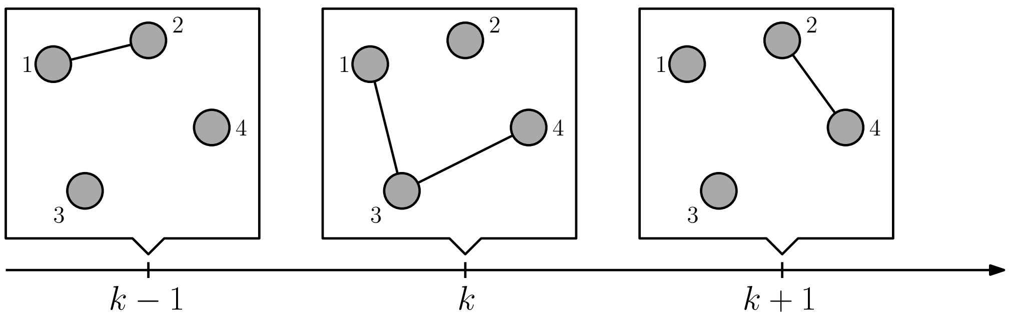


Figure 3.2. *An example of undirected time-varying graph with four devices.*

We can design a W_k which is still doubly stochastic, albeit non-connected, e.g., consider the Metropolis weights:

$$W_k^{ij} = \begin{cases} \frac{1}{1+\max\{d_k^i, d_k^j\}} & \text{for } (i,j) \in \mathcal{E}_k \\ 1 - \sum_j W_k^{ij} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases}$$

where here $\mathcal{E}_k$ is the edge set at time k. Then, we have the following result.

Theorem 3.3 *Consider the sequence of doubly stochastic matrices $\{W_k\}$. If there is an integer B, such that the union of the graphs,*

$$\bigcup_{k=t}^{t+B} \mathcal{G}_k, \qquad \forall t \in \mathbf{N}$$

is connected, then, for any $\boldsymbol{y}_0$ the recursion $\boldsymbol{y}_{k+1} = W_k \boldsymbol{y}_k$, converges linearly to

$$\lim_{k\to\infty} \boldsymbol{y}_k = \frac{\mathbf{1}\mathbf{1}^\top}{N} \boldsymbol{y}_0.$$

Proof. *See [NOR18].* ♣

The theorem says that if we consider an edge asynchronous setting, then we can still reach consensus to the average of the initial conditions. This is good news for the linear consensus iterations. This result can also extended to distributed optimization.

For DGD, we can look at the edge asynchronous setting with the master problem:

$$\min_{\boldsymbol{y}\in\mathbf{R}^{Nn}} \mathbf{E}_{\mathbf{W}}[\alpha F(\boldsymbol{y}) + \frac{1}{2}\boldsymbol{y}^\top (I - \mathbf{W})\boldsymbol{y}],$$

where now we are taking an expected value over the set of doubly stochastic matrices $\mathbf{W}$ and applying a stochastic gradient descent to it. This can be made to converge, albeit with errors as we know.

For Gradient tracking, with minor modifications (of the proof), we re-obtain linear convergence to zero error, but I spare you the details.

3.3 Node asynchronous graphs

We move now to the more realistic and harder setting: the setting in which the devices are the ones communicating asynchronously to their neighbors, and not the edges. This means that at every step k, the graphs will be directed. This is a much harder setting.

Consider Figure 3.3, then it is easy to see that for the recursion,

$$\boldsymbol{y}_{k+1} = \mathbf{W}_k \boldsymbol{y}_k,$$

the instantaneous matrix $\mathbf{W}_k$ cannot be made doubly stochastic, rendering reaching consensus harder to achieve. In particular, we only have either row or column stochastic, meaning $W_k\mathbf{1} = \mathbf{1}$ or $\mathbf{1}^\top W_k = \mathbf{1}^\top$. So either consensus or the average are preserved, but not both.

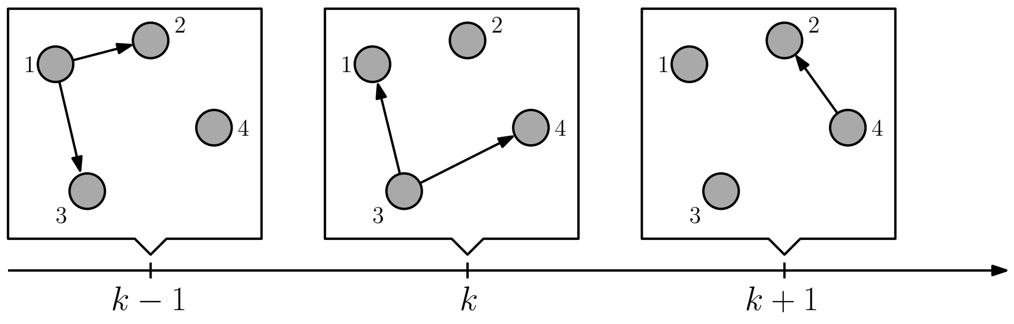


Figure 3.3. *An example of directed time-varying graph with four devices.*

3.3.1 The Push-sum protocol

We present now a communication strategy that can achieve consensus when dealing with a sequence of column stochastic matrices, the Push-Sum protocol.

Let us consider the following setting and protocol.

Push-sum

At time k:

- *Device j wakes up and sends two quantities to all its active neighbors: $\boldsymbol{x}_k^j/(1+d_{\text{out},k}^j) \in \mathbf{R}^N$ and supporting variable $\varphi_k^j/(1+d_{\text{out},k}^j) \in \mathbf{R}$. Here d_{out}^j is the outer degree: the number of neighbors device j communicates to.*
- *Each device i computes,*

$$\boldsymbol{x}_{k+1}^i = \sum_{j\in N_i^{\text{in}}\cup i} \frac{\boldsymbol{x}_k^j}{1+d_{\text{out},k}^j}, \quad \varphi_{k+1}^i = \sum_{j\in N_i^{\text{in}}\cup i} \frac{\varphi_k^j}{1+d_{\text{out},k}^j}, \quad z_{k+1}^i = \frac{\boldsymbol{x}_{k+1}^i}{\varphi_{k+1}^i},$$

here N_i^{in} is the set of neighbors that are communicating to i (incoming).

The Push-sum protocol can be written in matrix-vector form as,

$$\boldsymbol{y}_{k+1} = W_k \boldsymbol{y}_k, \quad \varphi_{k+1} = W_k \varphi_k, \quad z_{k+1}^i = \frac{\boldsymbol{x}_{k+1}^i}{\varphi_{k+1}^i}.$$

with W_k column stochastic. For example for Figure 3.3, then,

$$W_{k-1} = \begin{bmatrix} \frac{1}{3} & 0 & 0 & 0 \\ \frac{1}{3} & 1 & 0 & 0 \\ \frac{1}{3} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad W_k = \begin{bmatrix} 1 & 0 & \frac{1}{3} & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & \frac{1}{3} & 1 \end{bmatrix}, \quad W_{k+1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{bmatrix}$$

We have now the following result.

Theorem 3.4 (Push-sum) *Consider the sequence of column stochastic matrices $\{W_k\}$. If there is an integer B, such that the union of the graphs,*

$$\bigcup_{k=t}^{t+B} \mathcal{G}_k, \qquad \forall t \in \mathbf{N}$$

is strongly connected, then, for any $\boldsymbol{y}_0$ and $\varphi_0 = \mathbf{1}$, the Push-sum protocol converges as

$$\lim_{k\to\infty} z_k = \frac{\mathbf{1}\mathbf{1}^\top}{N} \boldsymbol{y}_0.$$

Proof. *See [NOR18].* ♣

In this harder setting, a lot of math is involved, but we can save most of the results we obtained, with a slow down in convergence rate due to the network.

3.4 Examples and beyond

Example 3.1 (Resit 2024) A faulty node

A number of devices communicate as in Figure 3.4. As you can see, the device labeled with 8 is faulty and it cannot send messages to devices 1 and 7, and it cannot receive anything from device 4. The devices want to solve the problem,

$$\min_{x \in \mathbf{R}^n} \sum_{i=1}^{N} \|x - y_i\|_2^2,$$

with $N = 8$ and with $y_i \in \mathbf{R}^n$ being a datum belonging to device i.

1. *Prove that the optimal solution is*

$$x^\star = \frac{1}{N} \sum_{i=1}^{N} y_i.$$

2. *Use the fact that you know that the optimizer is the average of the data, to derive a first-order distributed algorithm for the nodes to reach $x^\star$ based on the communication topology in Figure 3.4. How much communication you need between iterations? What is the convergence like?*

3. *Consider now a communication topology where we* remove *the directed communication links going from device 1 and device 7 to device 8 and from device 8 to device 4. Derive a first-order distributed algorithm for the nodes to reach $x^\star$ based on this new reduced communication topology. How much communication you need between iterations? What is the convergence like?*

4. *What could be a good algorithmic strategy if the communication topology would oscillate (change at every time) between the topology of Figure 3.4 and the one of Point 3? Argue and derive the algorithm.*

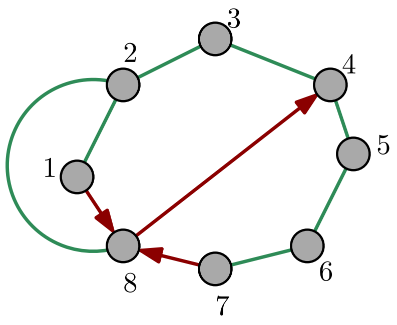


Figure 3.4. *A communication topology with a faulty node.*

Sketched solutions.

1. You can prove that the solution is the average of the data by writing the optimality conditions:

$$N x^* - \sum_{i=1}^{N} y_i = 0.$$

2. The communication graph is directed. To find the average, I can use the Push-sum algorithm. Based on the theory, I can say how many communications and what is the convergence like.

3. The communication graph is undirected. To find the average, a simple consensus protocol is enough. You can find all the properties in the theory.

4. If the topology oscillates between directed and undirected, you have no choice but to use a Push-sum algorithm, which is adapted to this case. This algorithm is however more complex than a simple consensus protocol. It may be wise, to just drop/ignore all the directed messages, and implement a simple consensus protocol, which is easier to code.

3.4.1 Application of Push-Sum to Distributed Optimization*

Research paper 6 *K. I. Tsianos, S. Lawlor and M. G. Rabbat,* Push-Sum Distributed Dual Averaging for Convex Optimization, *IEEE 51st IEEE Conference on Decision and Control (CDC), 2012*

3.4.2 Other gossip protocols*

Research paper 7 *Stephen Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah,* Randomized Gossip Algorithms, *IEEE Transactions on Information Theory, Vol. 52 (6), 2006*

Chapter 4

Distributed optimization: Dual methods

4.1 Introduction

We move now to dual methods, which have been developed to achieve a zero asymptotical error and allow for a much variety of assumptions and settings. Typically, they can handle constrained settings as unconstrained ones. This chapter follows mainly from [BPC+11, RB16].

4.1.1 Let's revisit our problem

In the previous chapters, we have discussed the philosophy of solving cooperative problems of the form,

$$\underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}} \sum_{i=1}^{N} f_i(\boldsymbol{x}),$$

by equipping each device with a copy of $\boldsymbol{x}$, say $\boldsymbol{x}^i$ and then forcing them to be equal by introducing a penalty term in the cost, for example $\frac{1}{2(\max_i\{d^i\}+1)}\sum_{i\sim j}\|\boldsymbol{x}^i-\boldsymbol{x}^j\|_2^2$.

A more direct way to obtain the same is to add the "forcing term" as a constraint in the optimization problem. The problem then becomes,

$$(\mathsf{Pu'}) \qquad \underset{\boldsymbol{x}^i\in\mathbf{R}^n, i=1,\dots,N}{\text{minimize}} \sum_{i=1}^{N} f_i(\boldsymbol{x}^i) \quad \text{subject to } \boldsymbol{x}^i=\boldsymbol{x}^j, \forall i\sim j, \tag{4.1}$$

where we recall that $i\sim j$ means that device i and j can communicate (i.e., there is a undirected edge between them).

We can then write (Pu') compactly as,

$$(\mathsf{Pu''}) \qquad \underset{\boldsymbol{y}\in\mathbf{R}^{Nn}}{\text{minimize}}\, F(\boldsymbol{y}) := \sum_{i=1}^{N} f_i(\boldsymbol{x}^i) \quad \text{subject to } A\boldsymbol{y}=0, \tag{4.2}$$

for variable $\boldsymbol{y}\in\mathbf{R}^{Nn}$ and carefully constructed matrix A.

Let us recall some definition and properties to have them at hand in this chapter. The variable $\boldsymbol{x}_k^i\in\mathbf{R}^n$ is the decision of device i at iteration k, and the stacked version is

$$\boldsymbol{y} = \begin{bmatrix}\boldsymbol{x}^1\\ \vdots \\ \boldsymbol{x}^N\end{bmatrix} \in \mathbf{R}^{Nn},$$

where a subscript k can be introduced if the variables refer to iteration k.

With the definition of F, if let f_i be L_i smooth and m_i strongly convex. Then,

$$f(\boldsymbol{x}) := \sum_{i=1}^{N} f_i(\boldsymbol{x}),$$

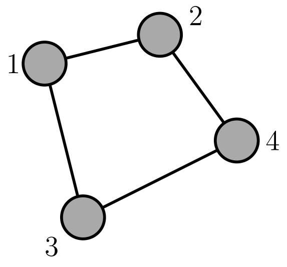
is L-smooth with $L = \sum_i L_i$ and $m = \sum_i m_i$. **However,** the function,

$$F(\boldsymbol{y}) = \sum_{i=1}^{N} f_i(\boldsymbol{x}^i),$$

is L'-smooth with $L' = \max_i\{L_i\}$ and $m' = \min_i\{m_i\}$. **Homework:** work it out.

Now, what is the matrix A in the definition of the constraints in (4.2)? This matrix does not have an unique form: depending on the connectivity, and the number of constraints, A can have different forms. Take the communication graph below among 4 devices as an example. Then, we can impose the constraints– and therefore A– as,

$$A = \begin{bmatrix} I_n & -I_n & 0 & 0 \\ 0 & I_n & -I_n & 0 \\ 0 & 0 & I_n & -I_n \\ I_n & 0 & 0 & -I_n \end{bmatrix} \in \mathbf{R}^{4n \times Nn}$$



Equivalently (since the last block row is redundant), we can impose the same constraints as,

$$A = \begin{bmatrix} -I_n & I_n & 0 & 0 \\ 0 & -I_n & I_n & 0 \\ 0 & 0 & -I_n & I_n \end{bmatrix} \in \mathbf{R}^{3n \times Nn}.$$

Another interesting (and less trivial) example is to set $A = \mathcal{L}_{\mathcal{G}} \otimes I_n$, the Laplacian of the graph, and so on and so forth.

Different ways to impose the constraints will have advantages and disadvantages. The first example of A has more constraints, so it is better for imposing them; however the resulting A is rank deficient, which may cause issues. The second example of A is full rank, but it has less constraints. We will see that there is no best strategy as we go along.

Now, to tackle problem (4.2), we will need to briefly revise duality in optimization.

4.2 Dual methods

4.2.1 Recap from OPT202

Most of the material in this subsection can be revised in the lecture notes of OPT202.

Consider the problem

$$\underset{\boldsymbol{x} \in \mathbf{R}^n}{\text{minimize}}\, f(\boldsymbol{x}) \quad \text{subject to } A\boldsymbol{x} = b, \tag{4.3}$$

for a convex function $f : \mathbf{R}^n \to \mathbf{R}$, and matrix $A \in \mathbf{R}^{p \times n}$, vector $b \in \mathbf{R}^p$ (with b in the image of A). The Lagrangian function is defined as,

$$\mathcal{L}(\boldsymbol{x}, \lambda) = f(\boldsymbol{x}) + \lambda^\top (A\boldsymbol{x} - b),$$

where $\lambda \in \mathbf{R}^p$ are the Lagrangian multipliers (also known as the dual variables). The Lagrangian dual function is then,

$$q(\lambda) = \inf_{\boldsymbol{x} \in \mathbf{R}^n} \mathcal{L}(\boldsymbol{x}, \lambda) = -(f^\star(-A^\top \lambda) + \lambda^\top b),$$

where $f^\star$ is the conjugate function of f, i.e., $f^\star(\boldsymbol{y}) = \sup_{\boldsymbol{x}}\{\boldsymbol{y}^\top x - f(\boldsymbol{x})\}$.

Homework. Prove the last equality.

The dual problem of (4.3) is then,

$$\max_{\lambda \in \mathbf{R}^p} q(\lambda) = \max_{\lambda \in \mathbf{R}^p} -(f^\star(-A^\top \lambda) + \lambda^\top b). \tag{4.4}$$

The dual problem is always convex, even if the primal is not and it provides a lower bound on the minimum of the primal problem, i.e.,

$$q^* \leqslant f^*.$$

Equality holds if some constraint qualification holds. For convex problems, if Slater's condition is verified (there exists a strictly feasible solution), then equality holds and we say that we have strong duality. For this course, strong duality always holds because we will not look at dualizing inequality constraints.

4.2.2 Our cooperation problem

With the intention of applying a dual method to our cooperation problem (4.2), we construct the Lagrangian as

$$\mathcal{L}(\boldsymbol{y}, \lambda) = F(\boldsymbol{y}) + \lambda^\top A\boldsymbol{y}$$

and its dual problem,

$$\max_{\lambda} \left[q(\lambda) := \inf_{\boldsymbol{y}}\{F(\boldsymbol{y}) + \lambda^\top A\boldsymbol{y}\} = -F^\star(-A^\top \lambda) \right]. \tag{4.5}$$

Equipped with it, can we now set up a subgradient descent for the dual problem, i.e., a dual ascent algorithm. Since the dual function may be non-differentiable, we recall the notion of subgradient as any element of the subdifferential at λ, i.e., $\partial q(\lambda)$.

Dual ascent algorithm

- *Start with λ_0 and a stepsize sequence $\{\alpha_k\}$*
- *Iterate for all $k = 0, 1, \ldots,$*

$$\lambda_{k+1} = \lambda_k + \alpha_k \boldsymbol{v}_k, \qquad \textit{with } \boldsymbol{v}_k \in \partial q(\lambda_k).$$

In the algorithm is important to note that the dual function may be non-differentiable, even if F is so, therefore we are forced to use the subgradient. Let us see what is the subgradient for our problem:

$$\partial_\lambda q(\lambda_k) \quad = \quad -A\partial_\lambda[-F^\star(-A^\top \lambda_k)].$$

Then, we know that for convex functions $(\partial_v F)^{-1}(v) = \partial_u F^\star(u)$, and in addition let,

$$\boldsymbol{y}^*(\lambda) := \arg\min_{\boldsymbol{y}} \mathcal{L}(\boldsymbol{y}, \lambda) \iff \partial F(\boldsymbol{y}^*(\lambda)) + A^\top \lambda \ni \mathbf{0} \iff$$
$$\boldsymbol{y}^*(\lambda) = (\partial F)^{-1}(-A^\top \lambda) = \partial F^\star(-A^\top \lambda)$$

Therefore, we have the relation

$$\partial q(\lambda_k) = A\boldsymbol{y}^*(\lambda_k),$$

that is: the subgradient is the residual map $A\boldsymbol{y}^*(\lambda_k)$.

4.3 Dual decomposition

4.3.1 Cloud-based dual decomposition

We are now ready for the first distributed dual method we will study: dual decomposition. It goes as follows.

Cloud-based dual decomposition

- *Start with a* λ_0 *and a stepsize sequence* $\{\alpha_k\}$
- *For* $k = 0, 1, \ldots$ *iterate:*
  - **Cloud communication step** *Send* λ_k *to all the devices*
  - **Per device computation step** *Solve:*

$$\boldsymbol{y}^*(\lambda) := \arg\min_{\boldsymbol{y}} \mathcal{L}(\boldsymbol{y}, \lambda_k) = \arg\min_{\boldsymbol{x}^1,\ldots,\boldsymbol{x}^N} \left\{ \sum_{i=1}^n f_i(\boldsymbol{x}^i) + \lambda_k^\top A[\boldsymbol{x}^1, \ldots, \boldsymbol{x}^N]^\top \right\}$$

which can be solved per device: upon dividing $A = [A_1|\ldots|A_N]$*, then,*

$$\boldsymbol{x}^{i,*}(\lambda_k) := \arg\min_{\boldsymbol{x}_i} \left\{ f_i(\boldsymbol{x}^i) + \lambda_k^\top A_i \boldsymbol{x}^i \right\} \qquad \forall i,$$

  - **Per device communication step** *Send to the cloud* $\boldsymbol{x}^{i,*}(\lambda_k)$
  - **Cloud computation step** *Compute the subgradient,*

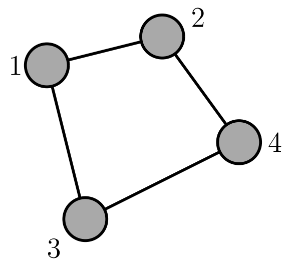
$$\partial q(\lambda_k) = A\boldsymbol{y}^*(\lambda_k) = \sum_i A_i \boldsymbol{x}^{i,*}(\lambda_k)$$

and update the multipliers:

$$\lambda_{k+1} = \lambda_k + \alpha_k \sum_i A_i \boldsymbol{x}^{i,*}(\lambda_k)$$

Example 4.1 *Let us fix the ideas of the method on a simple example. As we said, depending on the connectivity, and the number of constraints, A can be defined in different ways, for example,*

$$A = \left[\begin{array}{c|c|c|c} I_n & -I_n & 0 & 0 \\ 0 & I_n & -I_n & 0 \\ 0 & 0 & I_n & -I_n \\ I_n & 0 & 0 & -I_n \end{array}\right] =: [A_1|A_2|A_3|A_4]$$



In this case,

$$A\boldsymbol{y}^*(\lambda_k) = [A_1|A_2|A_3|A_4] \begin{bmatrix} \boldsymbol{x}^{1,*}(\lambda_k) \\ \boldsymbol{x}^{2,*}(\lambda_k) \\ \boldsymbol{x}^{3,*}(\lambda_k) \\ \boldsymbol{x}^{4,*}(\lambda_k) \end{bmatrix} = \sum_{i=1}^4 A_i \boldsymbol{x}^{i,*}(\lambda_k).$$

We report in Figure 4.1, a pictorial depiction of one iteration of the algorithm. The method is called dual decomposition, since you decompose the primal problem via the dual variables and you do not share the private costs f_i with the cloud. It involves two communication rounds and two computation rounds. The dual variables are updated globally, the primal locally, and this is convenient if f_i depends on data that you can't share and/or don't want to. Recall that $\lambda \in \mathbf{R}^{nC}$, where C are the number of constraints, so having less constraints may be good to keep the computation and communication low.

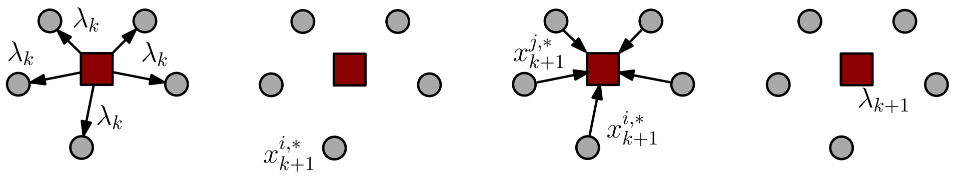


Figure 4.1. *The steps of Cloud-based dual decomposition per iteration k. The red square represents the cloud. We have let $x^{i,*}_{k+1} = \boldsymbol{x}^{i,*}(\lambda_k)$ for simplicity.*

4.3.2 Peer-to-peer dual decomposition

We look now at a version of dual decomposition that does not need the cloud: a peer-to-peer version. Here, we will try to let each device updates the λ's that belong to the constraints that it sees. For example, device i could update the λ's for $\boldsymbol{x}_i - \boldsymbol{x}_j = 0$, for all j it can communicate to. So, in this case, λ's live locally on the edges, and not on the cloud. Let's write this properly.

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, undirected. As usual, we indicate $i \sim j$ if there is an edge between i and j.

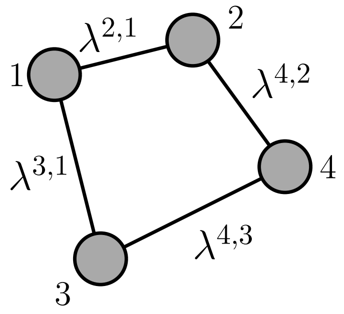


Figure 4.2. *The concept of $\lambda^{i,j}$ living on the edges for $j < i$ in an example.*

Let A be the matrix that collects the constraints $\boldsymbol{x}^i - \boldsymbol{x}^j$ for the edge set. Then the associated dual variable is λ^{ij} and that can live on the edge i, j. Let the number of edges be E, and we index the set of edges as,

$$\mathcal{E} = \{(i, j) \,|\, j < i, i \sim j\}.$$

Here, since the graph is undirected, there is no reason to count an edge twice, hence $j < i$.

As such $\boldsymbol{x}^i \in \mathbf{R}^n$, $A \in \mathbf{R}^{nE \times nN}$, if E edges and N nodes. Furthermore, λ^{ij} is associated to the edge of nodes i and j, and they can update it as,

$$\lambda^{ij}_{k+1} = \lambda^{ij}_k + \alpha_k[\boldsymbol{x}^{i,*}(\lambda_k) - \boldsymbol{x}^{j,*}(\lambda_k)], \qquad \forall k = 0, 1, \ldots$$

so we do not need a cloud. In fact, we can do the following:

Peer-to-peer dual decomposition

- *Initialize* $\lambda_0^{ij} \in \mathbf{R}^n$ *for* $(i,j) \in \mathcal{E}, j < i$, *and interpret* λ^{ij} *for* $j > i$ *as* λ^{ji}.
- *For* $k = 0, 1, \ldots$ *iterate:*
  - **Computation step** *For all devices* i *do,*

$$\boldsymbol{x}^{i,*}(\lambda_k) := \arg\min_{\boldsymbol{x}_i} \left\{ f_i(\boldsymbol{x}^i) + \sum_{i \sim j \equiv j \in N_i} (-1)^{a} \lambda_k^{ij,\top} \boldsymbol{x}^i \right\}$$

where

$$a = \begin{cases} 0 & \text{if } i > j \\ 1 & \text{otherwise} \end{cases}$$

  - **Communication step** *Send* $\boldsymbol{x}^{i,*}(\lambda_k)$ *to neighbors* j
  - **Computation step** *Update for node* i, j*:*

$$\lambda_{k+1}^{ij} = \lambda_k^{ij} + \alpha_k[\boldsymbol{x}^{i,*}(\lambda_k) - \boldsymbol{x}^{j,*}(\lambda_k)], \qquad j < i$$

We report in Figure 4.3, a pictorial depiction of one iteration of the algorithm. It involves one communication round and two computation rounds. The dual variables are updated locally as the primal, and we only share $\boldsymbol{x}^{i,*}$ with our neighbors.

4.3.3 Dual decomposition convergence

We look now at the convergence of dual decomposition. We need to recall some of the properties of the dual function q of our dual problem (4.4). In particular, let $\sigma(A)$ be the singular values of A.

1. If F is m-strongly convex then $-q$ is $\sigma^2_{\max}(A)/m$ smooth;
2. If F is L-smooth then $-q$ is $\sigma^2_{\min}(A)/L$ strongly convex.

Homework. Remind yourself of why it is so (check OPT202).

Then we have the following convergence guarantee.

Theorem 4.1 (Dual ascent convergence) *Consider problem (Pu") and the dual decomposition approach for a certain connection matrix A. If f_i's are m-strongly convex and L-smooth, then we can select $\alpha < 2m/\sigma^2_{\max}(A)$ and obtaining a linear rate of convergence for dual gradient ascent as,*

$$\|\lambda_k - \lambda^*\| \leqslant \left[\max\{|1 - \alpha\frac{\sigma^2_{\max}(A)}{m}|, |1 - \alpha\frac{\sigma^2_{\min}(A)}{L}|\}\right]^k \|\lambda_0 - \lambda^*\|$$

and

$$\|\boldsymbol{x}_k^i - \boldsymbol{x}^*\| \leqslant \frac{\sigma_{\max}(A)}{m}\|\lambda_k - \lambda^*\|,$$

where $(\boldsymbol{x}^*, \lambda^*)$ are the unique primal-dual optimizer of (Pu").

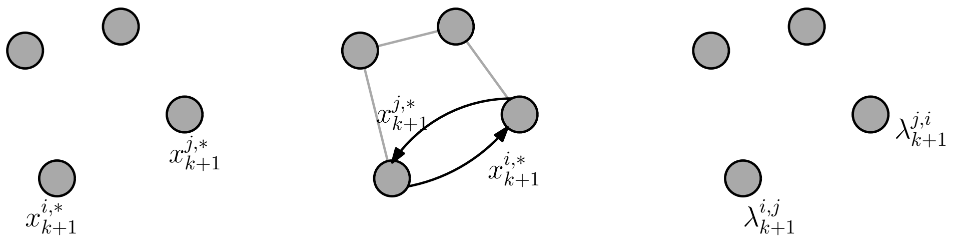


Figure 4.3. *The steps of Peer-to-peer dual decomposition per iteration k. We have let $x^{i,*}_{k+1} = \boldsymbol{x}^{i,*}(\lambda_k)$ for simplicity.*

Proof. *Proof follows from OPT201/202.* **Homework.** *Work it out: Be careful with the f_i's: they need to be strongly convex.* ♣

This theorem describes the requirements for dual decomposition to converge exactly to the primal-dual optimizer of the original problem (Pu"), thereby guaranteeing consensus and optimality. This is true when A is full row rank (so that $\sigma_{\min} > 0$). But why would it be? In fact, depending on the connectivity, and the number of constraints, A can be defined in different ways, some of which yields rank deficient A's. Let us see how to extend the theorem to cases in which A is not full rank.

4.3.4 Restricted strong convexity

To extend the theorem, we introduce the concept of restricted strong convexity. First of all, notice that in the case A is not full row rank, then the multipliers λ^* are not unique and $-q$ is not strongly convex.

However, upon decomposing λ^* as sum of two vector, one in the image of A and one in the null space of $A^\top$ (which we can always do), i.e., $\lambda^* = \lambda_0^* + \lambda_1^*$, with $\lambda_0^* \in \mathrm{im}(A)$ and $\lambda_1^* \in \mathrm{null}(A^\top)$, one can show that λ_0^* is unique and we concentrate on that one, since λ_1^* is redundant, given that $A^\top \lambda_1^* = 0$.

Start then with the multiplier $\lambda_0 \in \mathrm{im}(A)$; then, it is direct that:

$$\lambda_{k+1} = \lambda_k + \alpha A\boldsymbol{x} \in \mathrm{im}(A).$$

Definition 4.1 (Restricted strong convexity) *For a matrix A and a non-differentiable dual function $q(\lambda) : \mathbf{R}^n \to \mathbf{R}$, we say that $-q$ is restricted strongly convex with respect to matrix A iff, for any two vectors $\boldsymbol{v} \in \partial q(\lambda), \boldsymbol{v}' \in \partial q(\lambda')$:*

$$(\boldsymbol{v} - \boldsymbol{v}')^\top(\lambda' - \lambda) \geqslant \sigma_{\underline{\min}}^2/L\|\lambda' - \lambda\|^2, \quad \forall \lambda, \lambda' \in im(A),$$

with $\sigma_{\underline{\min}}$ the minimum non-zero singular value of A.

Can we now say that our $-q$ is restricted restricted strongly convex? Yes, substitute $\partial q(\lambda) = A\partial F^\star(-A^\top\lambda)$, then, for any vectors $\boldsymbol{v} \in \partial F^\star(-A^\top\lambda), \boldsymbol{v}' \in \partial F^\star(-A^\top\lambda')$:

$$(\boldsymbol{v} - \boldsymbol{v}')^\top A^\top(\lambda' - \lambda) = (\boldsymbol{v} - \boldsymbol{v}')^\top(-A^\top\lambda + A^\top\lambda') \geqslant$$
$$\text{steps from OPT202..!!} \geqslant 1/L\|A^\top(\lambda' - \lambda)\|^2 \geqslant \sigma_{\underline{\min}}^2/L\|\lambda' - \lambda\|^2.$$

The latest step is due to the fact that $A^\top(\lambda - \lambda') = 0$ iff $\lambda = \lambda'$.

So Theorem 4.1 becomes the following one.

Theorem 4.2 ((Full) Dual decomposition convergence) *Consider problem (Pu") and the dual decomposition approach for a certain connection matrix A. If f_i's are m-strongly convex and L-smooth, then we can select $\alpha < 2m/\sigma_{\max}^2(A)$ and $\lambda_0 \in im(A)$ and obtaining a linear rate of convergence for dual gradient ascent as,*

$$\|\lambda_k - \lambda^*\| \leqslant \left[\max\{|1 - \alpha\frac{\sigma_{\max}^2(A)}{m}|, |1 - \alpha\frac{\sigma_{\underline{\min}}^2(A)}{L}|\}\right]^k \|\lambda_0 - \lambda^*\|$$

and

$$\|\boldsymbol{x}_k^i - \boldsymbol{x}^*\| \leqslant \frac{\sigma_{\max}(A)}{m}\|\lambda_k - \lambda^*\|,$$

where $(\boldsymbol{x}^*, \lambda^*)$ are the unique primal - unique dual in the image of A optimizers of (Pu").

We remark here that choosing $\lambda_0 = \mathbf{0} \in \mathrm{im}(A)$, so this is an allowed choice. We also remark that λ_k is the vector whose components are the multipliers associated with the constraints in the row of A.

This theorem sums up all the properties of dual decomposition. Convergence is linear and to zero error, and it depends on the graph via the matrix A.

Example 4.2 (Laplacian constraint) *Upon setting $A = \mathcal{L}_\mathcal{G}$, which is an allowed choice, we can compute $\sigma_{\max}(A)$ and $\sigma_{\overline{\min}}(A)$ as $\lambda_N(\mathcal{L}_\mathcal{G}) \leqslant 2N$ and $\lambda_2(\mathcal{L}_\mathcal{G})$, respectively. Then, the best choice for α and the rate are,*

$$\alpha^* = \frac{2}{\frac{\sigma^2_{\max}(A)}{m} + \frac{\sigma^2_{\overline{\min}}(A)}{L}} \approx_{(N \gg 1, m \lesssim L)} \frac{L}{2N^2}, \quad \rho^* = \frac{\frac{\sigma^2_{\max}(A)}{m} - \frac{\sigma^2_{\overline{\min}}(A)}{L}}{\frac{\sigma^2_{\max}(A)}{m} + \frac{\sigma^2_{\overline{\min}}(A)}{L}} \approx_{(N \gg 1, m \lesssim L)} 1 - \frac{\lambda_2^2(\mathcal{L}_\mathcal{G})}{\lambda_N^2(\mathcal{L}_\mathcal{G})}.$$

meaning that for well-conditioned functions and loosely connected graphs, then the number of iterations one need to reach an accuracy ϵ is,

$$k \geqslant O\left(\frac{\lambda_N^2(\mathcal{L}_\mathcal{G})}{\lambda_2^2(\mathcal{L}_\mathcal{G})} \log(1/\epsilon)\right).$$

Sometimes, the ratio $\frac{\lambda_N(\mathcal{L}_\mathcal{G})}{\lambda_2(\mathcal{L}_\mathcal{G})}$ is also called spectral gap. For loosely connected graphs the ratio goes as $O(N)$ and we re-attain the $k \geqslant O(N^2 \log(1/\epsilon))$ dependency for convergence in general graphs. So dual decomposition is not "graph-wise faster" than Gradient Tracking, in general.

Convergence is trickier when F is not strongly convex, since q will not be differentiable, so it also hard to recover the primal solution here (see OPT202).

Convergence may be complicated when communication is directed, or asynchronous, or you have latencies, as in the first class.

4.3.5 Extensions to constrained problems

Dual decomposition can be modified to add decoupled local constraints as in Chapter 2.2.4. In particular, if $\boldsymbol{x}^i \in X_i \subseteq \mathbf{R}^n$, and we set

$$Y = X_1 \times X_2 \times \cdots \times X_n,$$

then each dual function can be computed as,

$$\boldsymbol{x}^{i,*}(\lambda_k) = \arg\min_{\boldsymbol{x}^i \in X_i} \{f_i(\boldsymbol{x}^i) + \lambda_k^\top(\boldsymbol{A}_i \boldsymbol{x}^i)\}.$$

Convergence in this case is not linear, since the dual function is not strongly concave. But possible, obtaining a $O(1/k^2)$-type convergence in the dual function by using Nesterov's acceleration (see OPT202 for details on the technique).

4.3.6 Numerical results for the Kernel example

We revisit here the Kernel example and show the convergence of peer-to-peer dual decomposition in Figure 4.4. As we can see, dual decomposition is not faster but it has the advantage of starting lower than Gradient Tracking, since we optimize the Lagrangian at each iteration.

For coding dual decomposition, it is useful to introduce some new data structure,

```
Edge_map = dict(), Edge_map[1] = (node_1, node_2)

Inverse_edge_map = dict(), Inverse_edge_map[(node_1, node_2)] = 1

Neighbors_map = dict(), Neighbors_map[1] = [2, 5]
```

4.4 The Alternating Direction Method of Multipliers

4.4.1 Introduction

We now look at one of the most versatile algorithms that is used in a variety of application scenarios, either centralized and decentralized, the Alternating Direction Method of Multipliers (ADMM). The method was first proposed in the 70's as a clever way to solve linear systems arising

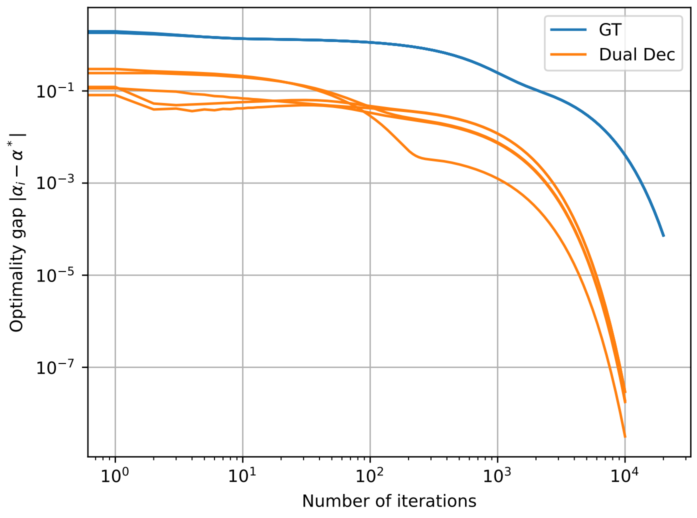


Figure 4.4. *Convergence in terms of $\|\alpha^i - \alpha^*\|$ for peer-to-peer dual decomposition and Gradient Tracking on the Kernel ridge regression example. Here I could choose the stepsize of dual decomposition as $s = 0.1$, since $m = 1/5$.*

from PDE discretization. It has gained "celebrity" for its weak convergence guarantees to solve optimization problems in the 2000's, and it is now employed even when it is not guaranteed to converge with surprising results (e.g., in combinatorial problems). A good account of the method is given in the survey [BPC+11], but you can find earlier accounts in [BT97].

As we have said, one of the main drawbacks of dual decomposition is that convergence is dictated by the choosing of a small stepsize, and that we are often limited by a strong convex cost. Along the years, researchers tried to add a regularization term, of the type, $+\|A\boldsymbol{x}\|_2^2$ to the cost, to weaken some of its assumptions. However, this regularization, which is a reminder of a multiplier method, is not distributable across the devices (the term $A^\top A$ couples devices that do not communication with each other).

To overcome this and other drawbacks of dual decomposition, we turn to ADMM. We are going to develop it for a general case and then focus it on our distributed problem

Consider the problem:

$$\min_{\boldsymbol{x}\in\mathbf{R}^n, \boldsymbol{y}\in\mathbf{R}^p} \quad f(\boldsymbol{x}) + g(\boldsymbol{y}) \tag{4.6}$$

$$\text{subject to} \quad A\boldsymbol{x} + B\boldsymbol{y} = c. \tag{4.7}$$

For well-defined matrices and vectors, as well as convex functions $f : \mathbf{R}^n \to \mathbf{R}, g : \mathbf{R}^n \to \mathbf{R}$.

Define the augmented Lagrangian,

$$\mathcal{L}_\beta(\boldsymbol{x}, \boldsymbol{y}, \lambda) := f(\boldsymbol{x}) + g(\boldsymbol{y}) + \lambda^\top(A\boldsymbol{x} + B\boldsymbol{y} - c) + \frac{\beta}{2}\|A\boldsymbol{x} + B\boldsymbol{y} - c\|^2,$$

for **any** scalar $\beta > 0$. Then the ADMM is the following algorithm.

ADMM (general case)

- *Start with an initial value for $\boldsymbol{x}_0, \boldsymbol{y}_0, \lambda_0$ and a positive scalar β,*
- *Iterate:*

$$\begin{aligned}
\boldsymbol{x}_{k+1} &= \arg\min_{\boldsymbol{x}\in\mathbf{R}^n} \mathcal{L}_\beta(\boldsymbol{x}, \boldsymbol{y}_k, \lambda_k) \\
\boldsymbol{y}_{k+1} &= \arg\min_{\boldsymbol{y}\in\mathbf{R}^p} \mathcal{L}_\beta(\boldsymbol{x}_{k+1}, \boldsymbol{y}, \lambda_k) \\
\lambda_{k+1} &= \lambda_{k+1} + \beta(A\boldsymbol{x}_{k+1} + B\boldsymbol{y}_{k+1} - c)
\end{aligned}$$

ADMM is similar to an incremental approach, but on two variables. It is also similar to dual decomposition, but on two variables. ADMM looks more complicated than dual gradient ascent: let us see if this complication leads to better guarantees.

In fact, if one assume strong duality holds and primal and dual solutions exist, then **convergence is ensured for any $\beta > 0$ in a weak sense.** This is a very strong result, which we do not prove here, but can be found in [BPC+11,RB16] and in later papers. So, ADMM is very powerful in general. Since, however, in the full generality, ADMM is also slow (with the lowest possible rate of $O(1/\sqrt{k})$), let us look at special cases.

4.4.2 ADMM convergence in a special case

Assume f to be m-strongly convex and L-smooth and that a primal-dual solution exists (We do not assume anything on g, just that it is a generic convex function). Assume that A is full row rank. Then ADMM converges **linearly** for any step size $\beta > 0$.

Theorem 4.3 (ADMM convergence (special case)) *Consider Problem (4.6) for $f \in \mathcal{S}^{1,1}_{m,L}(\mathbf{R}^n)$ and $g \in \Gamma(\mathbf{R}^p)$, and full rank matrix A. Assume that a primal-dual solution exists. Then, for any constant $\beta > 0$, the ADMM algorithm generates a sequence $\{\boldsymbol{x}_k, \boldsymbol{y}_k, \lambda_k\}$ which converges as,*

$$\|\lambda_k - \lambda^*\| \leqslant \varrho^k \|\lambda_0 - \lambda^*\| \qquad \varrho = \max\left\{\left|\frac{1-\beta\frac{\sigma^2_{\max}(A)}{m}}{1+\beta\frac{\sigma^2_{\max}(A)}{m}}\right|, \left|\frac{1-\beta\frac{\sigma^2_{\min}(A)}{L}}{1+\beta\frac{\sigma^2_{\min}(A)}{L}}\right|\right\}$$

and,

$$\|\boldsymbol{x}_k - \boldsymbol{x}^*\| \leqslant \frac{\sigma_{\max}(A)}{m}\|\lambda_k - \lambda^*\|.$$

Since ADMM is extremely popular, we have many ways to prove its convergence. Here, I want to give a glimpse of an operator-splitting technique (check OPT202 for the basis). The full proof can be found in [RB16].

Proof. *[Sketch] The proof goes as follows: ADMM is an application of a special algorithm (the Douglas-Rachford splitting) applied to the dual of our initial problem. The Douglas-Rachford splitting converges in a certain way given functional properties. We derive the dual of those functional properties and (as in the dual decomposition case) the convergence of the dual algorithm (ADMM). This is why the result looks very similar to the result of the dual decomposition. ADMM can be seen as a dual algorithm.*

Step I. Douglas-Rachford splitting. *Consider the problem,*

$$\min_{\boldsymbol{x}\in\mathbf{R}^n} f(\boldsymbol{x}) + g(\boldsymbol{x}),$$

with f and g convex closed and proper (CCP). Consider now the following method to find a solution $\boldsymbol{x}^*$.

Start at a certain z_0 and iterate for all $k \in \mathbb{N}$:

$$\boldsymbol{x}_k = \mathrm{prox}_{\beta f}(z_k) \tag{4.8a}$$

$$z_{k+1} = z_k + \mathrm{prox}_{\beta g}(2\boldsymbol{x}_k - z_k) - \boldsymbol{x}_k, \tag{4.8b}$$

where $\mathrm{prox}_{\beta\phi}$ is the usual prox operator:

$$\mathrm{prox}_{\beta\phi}(u) = \arg\min_v\{\phi(v) + \frac{1}{2\beta}\|v-u\|^2\}$$

The method is called the Douglas-Rachford splitting and it converges in some defined sense.

In particular if f is m-strongly convex and L-smooth, then, for all $\beta > 0$

$$\|z_{k+1} - z^*\| \leqslant \varrho^k\|z_k - z^*\|, \qquad \varrho = \max\left\{\left|\frac{1-\beta L}{1+\beta L}\right|, \left|\frac{1-\beta m}{1+\beta m}\right|\right\}$$

Step II. The dual problem and its properties *Start from our problem:*

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbf{R}^n,\boldsymbol{y}\in\mathbf{R}^p} \quad & f(\boldsymbol{x})+g(\boldsymbol{y}) \\ \textit{subject to} \quad & A\boldsymbol{x}+B\boldsymbol{y}=c, \end{aligned}$$

form the Lagrangian and take the dual

Dual problem is,

$$\max_{\lambda} \inf_{\boldsymbol{x},\boldsymbol{y}} \{f(\boldsymbol{x})+g(\boldsymbol{y})+\lambda^\top(A\boldsymbol{x}+B\boldsymbol{y}-c)\}$$

and so,

$$\min_{\lambda} f^\star(-A^\top\lambda)+g^\star(-A^\top\lambda)+c^\top\lambda$$

Consider "f"= $f^\star(-A^\top\lambda)+c^\top\lambda$ *and "g"=* $g^\star(-B^\top\lambda)$

As before we know that "f" is $\sigma^2_{\min}/L$*-strongly convex and* $\sigma^2_{\max}/m$*-smooth*

Step III. Applying DR to the dual *Apply DRS "f"=* $f^\star(-A^\top\lambda)+c^\top\lambda$ *and "g"=* $g^\star(-B^\top\lambda)$ *and use its convergence properties. Here we find the rate* ρ *of the Theorem.*

Step IV. Show that applying DR to the dual *amounts at the ADMM iterations.*

♣

4.5 Distributed ADMM

Let us go back to the problem,

$$\begin{aligned} \min_{\boldsymbol{x}\in\mathbf{R}^n,\boldsymbol{y}\in\mathbf{R}^p} \quad & f(\boldsymbol{x})+g(\boldsymbol{y}) \\ \text{subject to} \quad & A\boldsymbol{x}+B\boldsymbol{y}=c. \end{aligned}$$

We can use this problem template to model our cooperation problem,

$$\underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}} \sum_{i=1}^N f_i(\boldsymbol{x}).$$

For example, for a undirected and connected graph $\mathcal{G}=(\mathcal{V},\mathcal{E})$, we can write

$$\begin{aligned} \min_{\boldsymbol{x}^i\in\mathbf{R}^n,\boldsymbol{y}^{ij}\in\mathbf{R}^{|\mathcal{E}|}} \quad & \sum_{i=1}^N f_i(\boldsymbol{x}^i) \\ \text{subject to} \quad & \boldsymbol{x}^i=\boldsymbol{y}^{ij},\ \boldsymbol{x}^j=\boldsymbol{y}^{ij} \qquad \forall\, i\sim j. \end{aligned}$$

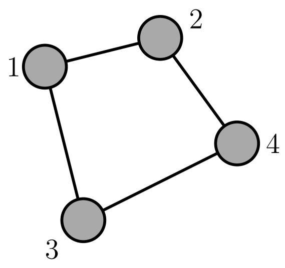
Think a bit at what is written. We have introduced supporting variables $\boldsymbol{y}^{ij}$'s, which live on the edges (interpret $\boldsymbol{y}^{ij}=\boldsymbol{y}^{ji}$ if $j>i$). The constraints impose that $\boldsymbol{x}^i=\boldsymbol{x}^j$ for all edges in a "contorted" way, which however gives us more flexibility in our assumptions!

This is of course not the only possibility. Different splittings between $\boldsymbol{x}$ and $\boldsymbol{y}$ and constraints yield different properties. This one will give a distributed algorithm similar to the one we have seen for dual decomposition. Later, we will see a different cloud-based splitting instead.

For the following, compactify $\boldsymbol{x}^i=\boldsymbol{y}^{ij}$ as, $A\boldsymbol{x}+B\boldsymbol{y}=0$. Here, the variables $\boldsymbol{x}^i\in\mathbf{R}^n, \boldsymbol{y}\in\mathbf{R}^{nE}$, but: $\lambda\in\mathbf{R}^{2nE}$, since there are two constraints per edge: $\boldsymbol{x}^i=\boldsymbol{y}^{ij}$, $\boldsymbol{x}^j=\boldsymbol{y}^{ij}$. Also, careful here that A is not full row rank, so linear convergence requires more work, but possible and we will see later how to obtain it.

Let us look at one example of A and B.

$$A = \begin{bmatrix} I_n & 0 & 0 & 0 \\ 0 & I_n & 0 & 0 \\ 0 & I_n & 0 & 0 \\ 0 & 0 & I_n & 0 \\ 0 & 0 & I_n & 0 \\ 0 & 0 & 0 & I_n \\ 0 & 0 & 0 & I_n \\ I_n & 0 & 0 & 0 \end{bmatrix}, \; B = -\begin{bmatrix} I_n & 0 & 0 & 0 \\ I_n & 0 & 0 & 0 \\ 0 & I_n & 0 & 0 \\ 0 & I_n & 0 & 0 \\ 0 & 0 & I_n & 0 \\ 0 & 0 & I_n & 0 \\ 0 & 0 & 0 & I_n \\ 0 & 0 & 0 & I_n \end{bmatrix}.$$



4.5.1 Peer-to-peer ADMM

We are now ready to write our peer-to-peer ADMM and simplify what can be simplified. Starting from the general algorithm, we can see that, at each iteration, the $\boldsymbol{x}$ step decouples among the devices, and each of them needs to update:

$$\boldsymbol{x}^i_{k+1} = \arg\min_{\boldsymbol{x}^i} \left\{ f_i(\boldsymbol{x}^i) + \sum_{j\sim i} \left((\lambda^{ij}_k)^\top (\boldsymbol{x}^i - \boldsymbol{y}^{ij}_k) + \frac{\beta}{2}\|\boldsymbol{x}^i - \boldsymbol{y}^{ij}_k\|^2 \right) \right\},$$

which is equivalent to,

$$\boldsymbol{x}^i_{k+1} = \arg\min_{\boldsymbol{x}^i} \left\{ f_i(\boldsymbol{x}^i) + \sum_{j\sim i} \frac{\beta}{2}\|\boldsymbol{x}^i - \boldsymbol{y}^{ij}_k + \frac{\lambda^{ij}_k}{\beta}\|^2 \right\}.$$

Recall that we interpret $\boldsymbol{y}^{ij} = \boldsymbol{y}^{ji}$ if $j > i$, but $\lambda^{ij} \neq \lambda^{ji}$.

The $\boldsymbol{y}$-step can also be written per device as the update,

$$\boldsymbol{y}^{ij}_{k+1} = \arg\min_{\boldsymbol{y}^{ij}} \left\{ (\lambda_k)^\top (A\boldsymbol{x}_{k+1} + B\boldsymbol{y}) + \frac{\beta}{2}\|A\boldsymbol{x}_{k+1} + B\boldsymbol{y}\|^2 \right\}$$

which is equivalent to,

$$\boldsymbol{y}^{ij}_{k+1} = \arg\min_{\boldsymbol{y}^{ij}} \left\{ \frac{\beta}{2}\|\boldsymbol{x}^i_{k+1} - \boldsymbol{y}^{ij} + \frac{\lambda^{ij}_k}{\beta}\|^2 + \frac{\beta}{2}\|\boldsymbol{x}^j_{k+1} - \boldsymbol{y}^{ij} + \frac{\lambda^{ij}_k}{\beta}\|^2 \right\}.$$

Solving for $\boldsymbol{y}^{ij}_{k+1}$ in this unconstrained problem gives,

$$\boldsymbol{y}^{ij}_{k+1} = \frac{\boldsymbol{x}^i_{k+1} + \boldsymbol{x}^j_{k+1}}{2} + \frac{\lambda^{ij}_k + \lambda^{ji}_k}{2\beta}.$$

Finally, the λ-step amounts at the update,

$$\lambda^{ij}_{k+1} = \lambda^{ij}_k + \beta(\boldsymbol{x}^i_{k+1} - \boldsymbol{y}^{ij}_{k+1}).$$

To simplify the equations, we notice that

$$\lambda^{ij}_{k+1} + \lambda^{ji}_{k+1} = \lambda^{ij}_k + \lambda^{ji}_k + \beta(\boldsymbol{x}^i_{k+1} + \boldsymbol{x}^k_{k+1} - 2\boldsymbol{y}^{ij}_{k+1}) = \boldsymbol{0}.$$

As such, $\boldsymbol{y}^{ij}_{k+1} = \frac{\boldsymbol{x}^i_{k+1} + \boldsymbol{x}^j_{k+1}}{2} + \frac{\lambda^{ij}_k + \lambda^{ji}_k}{2\beta} = \frac{\boldsymbol{x}^i_{k+1} + \boldsymbol{x}^j_{k+1}}{2}$.

With this in place, we can write the Peer-to-peer ADMM algorithm as follows.

Distributed (peer-to-peer) ADMM

- *Start with* $\lambda_0^{ij} = 0$, *and* $\boldsymbol{y}_0^{ij}$ *then,*
  - **Computation step** *Each device updates,*

$$\boldsymbol{x}_{k+1}^i = \arg\min_{\boldsymbol{x}^i} \left\{ f_i(\boldsymbol{x}^i) + \sum_{j\sim i} \frac{\beta}{2} \|\boldsymbol{x}^i - \boldsymbol{y}_k^{ij} + \frac{\lambda_k^{ij}}{\beta}\|^2 \right\}$$

  - **Communication step** *Each device communicates* $\boldsymbol{x}_{k+1}^i$ *to its neighbors*
  - **Computation step** *Each device updates,*

$$\boldsymbol{y}_{k+1}^{ij} = \frac{\boldsymbol{x}_{k+1}^i + \boldsymbol{x}_{k+1}^j}{2}$$

and,

$$\lambda_{k+1}^{ij} = \lambda_k^{ij} + \beta(\boldsymbol{x}_{k+1}^i - \boldsymbol{y}_{k+1}^{ij})$$

Peer-to-peer ADMM amounts at one communication and two computation steps per iteration.

4.5.2 Peer-to-peer ADMM convergence

Let us look now at its convergence guarantees. We do not look at the full proof, which is given in the research paper below, but at the intuition behind it. First, recall that we cannot apply Theorem 4.3 here, since A is not full rank. Second, notice that the problem of having a rank deficient A is similar to the same problem we had in dual decomposition. There we tackled the issue by using restricted strong convexity with respect to A. The problem with that is that λ in the ADMM depends on A and B, so it cannot remain in the image of A.

This is cleverly solved by noticing that in the Peer-to-peer ADMM updates, $\boldsymbol{y}$ is redundant and can be removed. In fact, we can set $\boldsymbol{y}_k = P\boldsymbol{x}_k$, for a carefully constructed matrix P (recall that $\boldsymbol{y}_k^{ij} = \frac{1}{2}(\boldsymbol{x}_k^i + \boldsymbol{x}_k^j)$). With this, then

$$\lambda_{k+1} = \lambda_k + \beta(A + BP)\boldsymbol{x}_{k+1}.$$

So if λ_0 is initialized in the image of $A + BP$, λ_k lives there too. We then search for restricted strong convexity with respect to $A+BP$ and prove a similar linear convergence result. Depending on A, B the proof may be more complex. In the research paper below, one can see one possible "easy" choice for A, B and the initialization for $\lambda_0 = \mathbf{0}$, which guarantee linear convergence of Peer-to-peer ADMM, also for rank deficient A's. The paper also reports how to select the optimal β to optimize the rate.

Research paper 8 *W. Shi, Q. Ling, K. Yuan, G. Wu and W. Yin,* On the Linear Convergence of the ADMM in Decentralized Consensus Optimization, *in IEEE Transactions on Signal Processing, vol. 62, no. 7, pp. 1750-1761, 2014*

4.5.3 Extensions to constrained problems

ADMM can be modified to add decoupled local constraints as in Chapter 2.2.4. In particular, if $\boldsymbol{x}^i \in X_i \subseteq \mathbf{R}^n$, and we set

$$Y = X_1 \times X_2 \times \cdots \times X_n,$$

then each $\boldsymbol{x}$-step can be modified by optimizing over $\boldsymbol{x}^i \in X_i$. Convergence in this case is not linear.

4.6 Numerical results

4.6.1 Kernel setting

We revisit once more our kernel regression setting and apply ADMM to it. Here, we can choose whichever stepsize $\beta > 0$ we want. The sequence of subfigures in Figure 4.5 demonstrate that

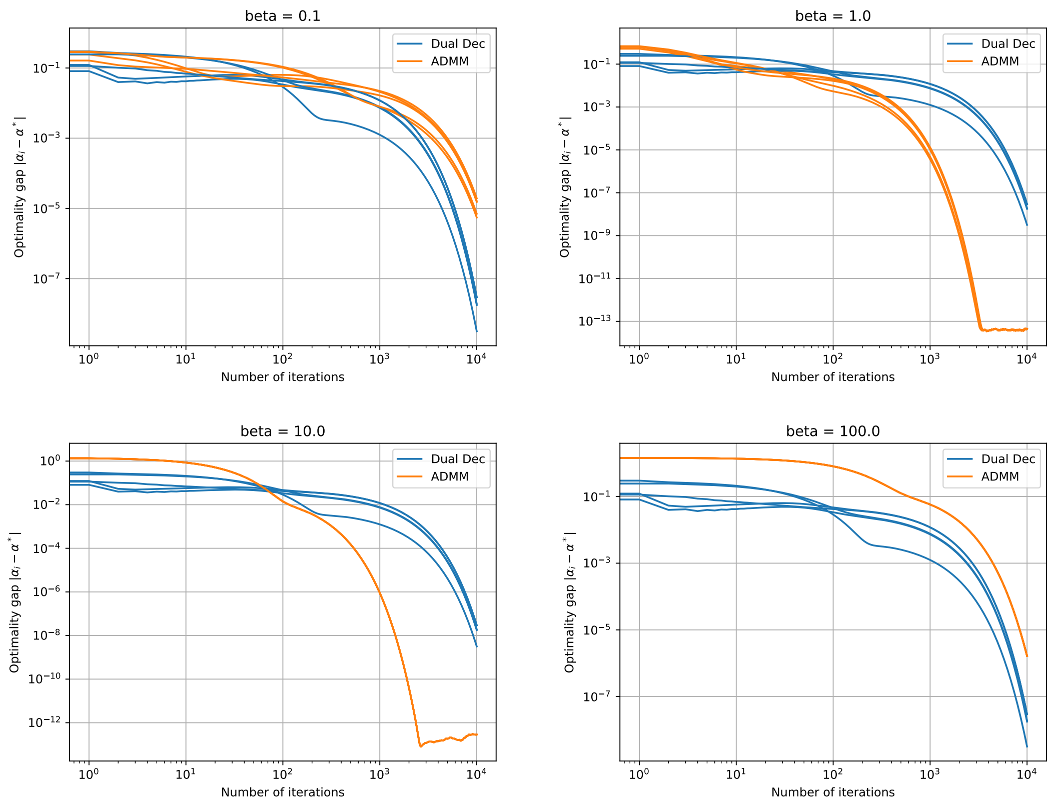


Figure 4.5. .

ADMM is superior to dual decomposition if we select β carefully.

4.7 ADMM examples

We look now at some interesting optimization and machine learning examples for ADMM and at its cloud-based formulation.

4.7.1 Model fitting

Example 4.3 (Model fitting) *Consider the task of training a model via the convex problem,*

$$\min_{\boldsymbol{x}\in\mathbf{R}^n} \ell(A\boldsymbol{x}-b)+r(\boldsymbol{x}),$$

where $\ell:\mathbf{R}^p\to\mathbf{R}$ is a convex loss function, $A\in\mathbf{R}^{p\times n}$ is the feature matrix, $b\in\mathbf{R}^p$ is the output vector, $\boldsymbol{x}$ are the parameters of the model, and $r:\mathbf{R}^n\to\mathbf{R}$ is a convex regularization function. Assume that ℓ is additive across training examples, as

$$\ell(\boldsymbol{x})=\sum_{i=1}^m \ell_i(a_i^\top\boldsymbol{x}-b_i).$$

Assume also that r is also separable. This is rather typical: for instance if r is the Tikhonov regularization (aka ridge penalty): $r(\boldsymbol{x})=\nu\|\boldsymbol{x}\|_2^2$, or if r is the lasso penalty: $r(\boldsymbol{x})=\nu\|\boldsymbol{x}\|_1$. Both of which are separable across the devices.

In machine learning, typically you have a modest number of features but a very large number of training examples (i.e., $m\gg n$), so it makes sense to try to solve the problem in a distributed way, with each processor handling a subset of the training data. This is useful either when there are so many training examples that it is inconvenient or impossible to process them on a single machine or when the data is naturally collected or stored in a distributed fashion. This includes, *for example, online social network data, webserver access logs, wireless sensor networks, and many cloud computing applications more generally.*

Suppose you associate a number of training examples to a number N of processors (in the extreme $N = m$). Then, you may solve

$$\begin{aligned}\min_{\boldsymbol{x}^j\in\mathbf{R}^n,\boldsymbol{y}\in\mathbf{R}^n} \quad & \sum_{j=1}^N \ell_j(A_j\boldsymbol{x}^j - b_j) + r(\boldsymbol{y}),\\ \textit{subject to} \quad & \boldsymbol{x}^j = \boldsymbol{y}, \quad i = 1,\dots,N.\end{aligned}$$

This can be solved via ADMM.

- *Each processor solves*

$$\begin{aligned}\boldsymbol{x}^j_{k+1} = \arg\min_{\boldsymbol{x}^j}\{\ell_j(A_j\boldsymbol{x}^j - b^j) + \lambda^j_k(\boldsymbol{x}^j - \boldsymbol{y}_k) + \frac{\beta}{2}\|\boldsymbol{x}^j - \boldsymbol{y}_k\|^2\}\\ = \arg\min_{\boldsymbol{x}^j}\{\ell_j(A_j\boldsymbol{x}^j - b^j) + \frac{\beta}{2}\|\boldsymbol{x}^j - \boldsymbol{y}_k + \frac{\lambda^j_k}{\beta}\|^2\}\end{aligned}$$

- *Communication to the cloud of $\boldsymbol{x}^j_{k+1}$*
- *On the cloud we solve,*

$$\begin{aligned}\boldsymbol{y}_{k+1} = \arg\min_{\boldsymbol{y}}\{r(\boldsymbol{y}) + \sum_{j=1}^N \lambda^j_k(\boldsymbol{x}^j_{k+1} - \boldsymbol{y}) + \frac{\beta}{2}\|\boldsymbol{x}^j_{k+1} - \boldsymbol{y}\|^2\}\\ = \arg\min_{\boldsymbol{y}}\{r(\boldsymbol{y}) + \sum_{j=1}^N \frac{\beta}{2}\|\boldsymbol{x}^j_{k+1} - \boldsymbol{y} - \frac{\lambda^j_k}{\beta}\|^2\}\end{aligned}$$

- *Update $\lambda^j_{k+1} = \lambda^j_k + \beta(\boldsymbol{x}^j_{k+1} - \boldsymbol{y}_{k+1})$ and communicate back to processors $\boldsymbol{y}_{k+1}, \lambda^j_{k+1}$*

The data never leaves the processors. Different updates of $\boldsymbol{y}$ depend on r, and of $\boldsymbol{x}$ depend on ℓ. Consider for example: $\ell(\cdot) = \|\cdot\|_2^2$, $r(\boldsymbol{y}) = \nu\|\boldsymbol{y}\|_1$, then we obtain the algorithm,

- *Each processor solves*

$$\begin{aligned}\boldsymbol{x}^j_{k+1} = \arg\min_{\boldsymbol{x}^j}\{\|A_j\boldsymbol{x}^j - b_j\|^2 + \frac{\beta}{2}\|\boldsymbol{x}^j - \boldsymbol{y}_k + \frac{\lambda^j_k}{\beta}\|^2\}\\ = (A_j^\top A_j + \beta I)^{-1}(A_j^\top b_j + \beta\boldsymbol{y}_k - \lambda^j_k).\end{aligned}$$

- *Communication to the cloud of $\boldsymbol{x}^j_{k+1}$*
- *On the cloud we solve,*

$$\boldsymbol{y}_{k+1} = \arg\min_{\boldsymbol{y}}\{\nu\|\boldsymbol{y}\|_1 + \sum_{j=1}^N \frac{\beta}{2}\|\boldsymbol{x}^j_{k+1} - \boldsymbol{y} + \frac{\lambda^j_k}{\beta}\|^2\} = S_{\nu/\beta}(\bar{\boldsymbol{x}}_{k+1} + \frac{\bar{\lambda}_k}{\beta})$$

- *Update $\lambda^j_{k+1} = \lambda^j_k + \beta(\boldsymbol{x}^j_{k+1} - \boldsymbol{y}_{k+1})$ and communicate back to processors $\boldsymbol{y}_{k+1}, \lambda^j_{k+1}$*

4.7.2 Further examples from Exams.

Example 4.4 (Exam 2024) *We reconsider Example 2.2.*

4. *Consider now using the ADMM algorithm on the original problem. Derive a peer-to-peer ADMM algorithm that can solve the problem at optimality. Describe how many communication rounds per iteration you need and what you exchange.*

5. *Re-consider now using the ADMM algorithm, but now in a cloud-based setting. Let $w \in \mathbf{R}^n$ be the global decision variable vector and model the problem as solving,*

$$\min_{x_i \in \mathbf{R}^{n_i}, i=1,\ldots,N} \quad \frac{1}{2}\sum_{i=1}^{N} \|y_i - C_i x_i\|_2^2 + \nu\|w\|_1, \qquad \textit{subject to: } x_i = U^i w, \quad \forall i,$$

where $U^i \in \mathbf{R}^{n_i \times n}$ is a selection matrix that tells which part of w is x_i. Here we have added a regularization $\nu\|w\|_1$, with $\nu > 0$.

Derive a cloud-based ADMM to solve the above problem, in which the sensors communicate all to a server. Describe how many communication rounds per iteration you need and what you exchange. What about convergence?

Sketched solutions

4. We recall the problem,

$$\min_{x_i \in \mathbf{R}^{n_i}, i=1,\ldots,N} \quad \frac{1}{2}\sum_{i=1}^{N} \|y_i - C_i x_i\|_2^2, \qquad \text{subject to: } V^{ij}x_i = V^{ji}x_j, \quad \forall i \sim j,$$

We can set up a peer-to-peer ADMM by considering the splitting,

$$\min_{x_i \in \mathbf{R}^{n_i}, i=1,\ldots,N, z_{ij}} \quad \frac{1}{2}\sum_{i=1}^{N} \|y_i - C_i x_i\|_2^2, \qquad \text{subject to: } \{\; V^{ij}x_i = z_{ij}, V^{ji}x_j = z_{ij} \qquad \forall i \sim j,$$

Then we can write the ADMM algorithm as

- Each device solves,

$$x_i^{k+1} = \arg\min_{x_i}\{\frac{1}{2}\|y_i - C_i x_i\|_2^2 + \left[\lambda_k^{ij,\top}(V^{ij}x_i - z_{ij}^k) + \frac{\beta}{2}\|V^{ij}x_i - z_{ij}^k\|^2\right]\}$$

  and sends x_i^{k+1} to the neighbors,

- Each device updates the supporting variables,

$$z_{ij}^{k+1} = \frac{V^{ij}x_i^{k+1} + V^{ji}x_j^{k+1}}{2}, \qquad \lambda_{k+1}^{ij} = \lambda_k^{ij} + \beta(V^{ij}x_i^{k+1} - z_{ij}^{k+1})$$

Then you can describe how many communication rounds and what you send.

5. Similar as before, the ADMM is,

- Each device solves,

$$x_i^{k+1} = \arg\min_{x_i}\{\frac{1}{2}\|y_i - C_i x_i\|_2^2 + \sum_{i\sim j}\left[\lambda_k^{i,\top}(x_i - U^i w_k) + \frac{\beta}{2}\|x_i - U^i w_k\|^2\right]\}$$

  and sends x_i^{k+1} to the cloud,

- The cloud updates the supporting variables,

$$w^{k+1} = \arg\min_{w}\{\nu\|w\| + \sum_{i=1}^{N}[\lambda_k^{i,\top}(x_i^{k+1} - U^i w) + \frac{\beta}{2}\|x_i^{k+1} - U^i w\|^2]\},$$

$$\lambda_{k+1}^{i} = \lambda_k^{i} + \beta(x_i^{k+1} - U^i w^{k+1})$$

  and sends them to the devices.

Since the cost is not strongly convex in general, we cannot use the Theorems we have seen in class, but ADMM will converge nonetheless (if you know how, great, otherwise this simple statement is sufficient).

4.8 Asynchronous ADMM *

Research paper 9 *Iutzeler, Franck, Pascal Bianchi, Philippe Ciblat, and Walid Hachem.* Asynchronous distributed optimization using a randomized alternating direction method of multipliers. *In 52nd IEEE conference on decision and control, pp. 3671-3676. IEEE, 2013.*

4.9 Alternative methods

4.9.1 Distributed dual averaging*

Research paper 10 *Duchi, John C., Alekh Agarwal, and Martin J. Wainwright.* Dual averaging for distributed optimization: Convergence analysis and network scaling. *IEEE Transactions on Automatic control 57 (3), 2011*

4.9.2 Optimal convergence rates*

Research paper 11 *Scaman, Kevin, Francis Bach, Sébastien Bubeck, Yin Tat Lee, and Laurent Massoulié.* Optimal convergence rates for convex distributed optimization in networks. *Journal of Machine Learning Research 20, 2019*

Chapter 5

Constraint-coupled problems

5.1 Introduction

In this chapter, we move our attention to constrained-coupled problems, that is cooperation problems of the form,

$$\underset{\boldsymbol{x}^i \in \mathbf{R}^n, i=1,\ldots,N}{\text{minimize}} \quad \sum_{i=1}^N f_i(\boldsymbol{x}^i), \text{ subject to } \sum_{i=1}^N A_i \boldsymbol{x}^i = b, \sum_{i=1}^N g_i(\boldsymbol{x}^i) \leqslant 0, \tag{5.1}$$

for convex functions $f_i : \mathbf{R}^n \to \mathbf{R}$, $g_i : \mathbf{R}^n \to \mathbf{R}^m$, matrices $A_i \in \mathbf{R}^{p \times n}$ and vector $b \in \mathbf{R}^p$. We notice that the cost is decoupled, i.e., each device has its own decision variable $\boldsymbol{x}^i$ and cost f_i. However the constraints combine multiple decision variables in a non-trivial way. A notable example of such problems is when we want to allocate a fixed budget to different devices. If x^i represents the energy device i has, we could think of solving a problem where we fix the total energy budget as the constraint,

$$\sum_{i=1}^N x^i = B.$$

A possible way to tackle the constrained-coupled problem (5.1) in a cooperative way is to use duality and convert it into a cost-coupled problem in the dual variables. Indeed, the Lagrangian dual problem of (5.1) is,

$$\underset{\boldsymbol{y} \in Y \subseteq \mathbf{R}^{p+m}}{\text{minimize}} \sum_{i=1}^N - \left[q_i(\boldsymbol{y}) := \inf_{\boldsymbol{x}^i \in \mathbf{R}^n} f_i(\boldsymbol{x}^i) + \lambda^\top (A_i \boldsymbol{x}^i - b/N) + \nu^\top (g_i(\boldsymbol{x}^i)) \right], \tag{5.2}$$

where we have set $\boldsymbol{y} = [\lambda^\top, \nu^\top]^\top$. This is our usual cooperation problem, but now in the dual variables. We will see below a few ways to characterize this dual problem and alternative formulations.

5.2 Primal and dual decomposition*

Research paper 12 *Boyd, Stephen, Lin Xiao, Almir Mutapcic, and Jacob Mattingley.* Notes on decomposition methods. *Notes for EE364B, Stanford University 635, 2007*

5.3 Resource allocation*

Research paper 13 *Xiao, Lin, and Stephen Boyd.* Optimal scaling of a gradient method for distributed resource allocation. *Journal of optimization theory and applications 129, 2006*

5.4 Consensus-based dual decomposition*

Research paper 14 *Simonetto, Andrea, and Hadi Jamali-Rad.* Primal recovery from consensus-based dual decomposition for distributed convex optimization. *Journal of Optimization Theory and Applications 168, 2016*

5.5 Tracking ADMM*

Research paper 15 *Falsone, Alessandro, Ivano Notarnicola, Giuseppe Notarstefano, and Maria Prandini.* Tracking-ADMM for distributed constraint-coupled optimization. *Automatica 117, 2020*

Chapter 6

Stochastic optimization and variance reduction

6.1 Introduction

Before moving on to cooperative learning problems, it is useful to introduce (or recall) some fundamental results in stochastic optimization. We will see that this area is closely related to distributed optimization and learning and it will be our link from the former to the latter. I am not aiming here at giving a full account of the development of machine learning, just a few pointers to tools and results that we will use later on.

Consider a parametric convex function $f : \mathbf{R}^n \times \mathbf{R}^p \to \mathbf{R}$, which is convex in $\boldsymbol{x} \in \mathbf{R}^n$ for all the values of a parameter $\theta \in \mathbf{R}^p$. We write f as $f(\boldsymbol{x}; \theta)$. For us f is a cost function parametrized by some noise vector θ. For example, we can have,

$$f(\boldsymbol{x}; \theta) = \frac{1}{2}\|A\boldsymbol{x} - b - \bar{y} + \theta\|^2,$$

if we are trying to fit a linear model $A\boldsymbol{x} - b$ with features $\boldsymbol{x}$ to some noisy observations $\bar{y} + \theta$. For us θ will be a random vector drawn from a known distribution Θ and we will write compactly $\theta \sim \Theta$.

We will also define the expected value operator $\mathbf{E}$ in the usual way as,

$$\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x}; \theta)] := \int_{\theta\in\Theta} f(\boldsymbol{x}; \theta) p(\theta) \mathrm{d}\theta.$$

Then, we look at the stochastic optimization problem,

$$\textsf{(EP)} \qquad \underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}}\, \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x}; \theta)]. \tag{6.1}$$

with a convex function $f : \mathbf{R}^n \times \mathbf{R}^p \to \mathbf{R}$ in $\boldsymbol{x}$ and noise $\theta \sim \Theta$. This is a stochastic optimization problem often encountered in machine learning. Sometimes it is referred to as *risk minimization* problem.

We recall that $\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x}; \theta)]$ is a convex function of $\boldsymbol{x}$, since f is convex in $\boldsymbol{x}$, and $\mathbf{E}$ is convex combination of convex functions, so (EP) is a convex problem.

We can ask immediately the question: is (EP) very different from our cooperative problem,

$$\textsf{(P)} \qquad \min_{\boldsymbol{x}\in\mathbf{R}^n} \sum_{i=1}^N f_i(\boldsymbol{x}) \equiv \frac{1}{N}\sum_{i=1}^N f_i(\boldsymbol{x})\,?$$

In fact, we can think of (EP) as,

$$\textsf{(EP)} \qquad \min_{\boldsymbol{x}\in\mathbf{R}^n} \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x}; \theta)] = \lim_{N\to\infty} \frac{1}{N}\sum_{i=1}^N f_i(\boldsymbol{x}),$$

where each $f_i = f(\boldsymbol{x}; \theta_i)$. So, solving (EP) can't be very different from solving (P). In fact, in most machine learning problems, N is anyway finite, since represents the data samples one has. In this context, we will consider the data-oriented *empirical* loss,

$$\text{(EDP)} \qquad \min_{\boldsymbol{x} \in \mathbf{R}^n} \frac{1}{N} \sum_{i=1}^{N} f_i(\boldsymbol{x}), \tag{6.2}$$

where N is the number of samples one has.

Remark 1 *Notice however already an important difference between (EP) or (EDP) and (P): in the latter f_i can be anything, while here $f_i(\boldsymbol{x}) = f(\boldsymbol{x}; \theta_i)$. In addition, N are samples in the former, and the number of devices in the latter.*

6.2 Stochastic gradient descent

6.2.1 The basics

The first method we present is the celebrated stochastic gradient descent (SGD), which is workhorse of training machine learning models (with some of its fancier variants). We will drop the subscript $\theta \sim \Theta$ from $\mathbf{E}$, to ease notation, whenever clear by the context.

We will need to change nomenclature somewhat, since moving from optimization to machine learning. The following list may help you navigate the next few chapters.

Optimization		Machine Learning
Cost function	$\Longleftrightarrow$	Loss function
Stepsize	$\Longleftrightarrow$	Learning rate
Number of iterations	$\Longleftrightarrow$	Number of epochs

The idea behind SGD is the idea of the incremental gradient that we have seen in Chapter 1.3.1: use $\nabla f_i(\boldsymbol{x})$ as a proxy for the real gradient $\nabla_{\boldsymbol{x}} \mathbf{E}[f(\boldsymbol{x}; \theta)]$. This makes sense, since, by linearity of the expectation operator, $\mathbf{E}[\cdot]$, the gradient of the cost with respect to $\boldsymbol{x}$ is

$$\nabla_{\boldsymbol{x}} \mathbf{E}[f(\boldsymbol{x}; \theta)] = \mathbf{E}[\nabla_{\boldsymbol{x}} f(\boldsymbol{x}; \theta)] = \lim_{N \to \infty} \frac{1}{N} \sum_{i=1}^{N} \nabla f_i(\boldsymbol{x})$$

so that each $\nabla f_i(\boldsymbol{x})$ is an **unbiased estimator** of $\nabla_{\boldsymbol{x}} \mathbf{E}[f(\boldsymbol{x}; \theta)]$.

We will concentrate here on the empirical minimization problem (6.2). Looking back at the incremental gradient method, we can already derive SGD as follows.

Stochastic Gradient Descent (SGD)

- *Start with a $\boldsymbol{x}_0 \in \mathbf{R}^n$, a learning rate sequence $\{\alpha_k\}$, and a finite number of samples N*
- *Iterate for all epochs $k = 0, 1, \ldots$: pick at random sample i,*

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f_i(\boldsymbol{x}_k)$$

6.2.2 SGD convergence

SGD convergence goes as follows (in par with Theorem 1.2).

Theorem 6.1 (SGD convergence with strong convexity) *Consider the problem (EDP) and its optimizer $\boldsymbol{x}^*$. Assume a bounded variance with respect to the optimum, meaning,*

$$\mathbf{E}[\|\nabla f_i(\boldsymbol{x}) - \frac{1}{N} \sum_{j=1}^{N} \nabla f_j(\boldsymbol{x}^*)\|^2] = \mathbf{E}[\|\nabla f_i(\boldsymbol{x})\|^2] \leqslant G^2,$$

for a finite constant $G > 0$.

Furthermore, assume m-strong convexity and differentiability of f with respect to $\boldsymbol{x}$.

Then, SGD with a constant learning rate $\alpha \leqslant 1/(2m)$ will generate a sequence $\{\boldsymbol{x}_k\}$ which converges as,

$$\mathbf{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] \leqslant (1-2m\alpha)^k \mathbf{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2] + \frac{\alpha}{2m} G^2.$$

Moreover, if one selects a vanishing learning rate decaying as $\alpha_k = 1/(2mk)$, then,

$$\lim_{k\to\infty} \mathbf{E}[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] = 0$$

Proof. *The proof of the theorem is quite standard. Start with the update rule,*

$$\begin{aligned} \|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2 &= \|\boldsymbol{x}_k - \alpha \nabla f_i(\boldsymbol{x}_k) - \boldsymbol{x}^*\|^2 \\ &= \|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 - 2\alpha(\nabla f_i(\boldsymbol{x}_k))^\top(\boldsymbol{x}_k - \boldsymbol{x}^*) + \alpha^2 \|\nabla f_i(\boldsymbol{x}_k)\|^2. \end{aligned}$$

Taking expectation with respect to i, since $\mathbf{E}_i[\nabla f_i(\boldsymbol{x}_k)] = \nabla f(\boldsymbol{x}_k)$, and recalling strong convexity, $(\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}^*))^\top (\boldsymbol{x}_k - \boldsymbol{x}^*) \geqslant m\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2$, we have

$$\mathbf{E}_i[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] \leqslant (1-2m\alpha)\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 + \alpha^2 G^2.$$

Taking total expectation,

$$\mathbf{E}[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] \leqslant (1-2m\alpha)\mathbf{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] + \alpha^2 G^2.$$

By geometric series, the first part of the theorem is then proven.

Now, take $\alpha_k = \frac{1}{2mk}$, then, the previous equation reads,

$$\begin{aligned} \mathbf{E}[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] &\leqslant (1-1/k)\mathbf{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] + \left(\frac{G}{2mk}\right)^2 = \left(\prod_{t=2}^{k} \frac{t-1}{t}\right)\mathbf{E}[\|\boldsymbol{x}_2 - \boldsymbol{x}^*\|^2] \\ &+ \left(\frac{G}{2mk}\right)^2 + \frac{k-1}{k}\left(\frac{G}{2m(k-1)}\right)^2 + \ldots + \left(\prod_{t=2}^{k} \frac{t-1}{t}\right)\left(\frac{G}{2m}\right)^2 = \\ &= \frac{1}{k}\mathbf{E}[\|\boldsymbol{x}_2 - \boldsymbol{x}^*\|^2] + \left(\frac{G}{2m}\right)^2 \frac{1}{k}\sum_{i=1}^{k}\frac{1}{i} \leqslant O(\log(k)/k). \end{aligned}$$

Taking the limit, the second part of the theorem is also proven. ♣

Theorem 6.1 depicts again trade-off between speed and error. You can go linearly to an error bound or slowly to zero error. This is typical for SGD, even for fancier versions and proof techniques, as you can see in [GLQ$^+$19, GG23]. But, we can already notice that we have not used smoothness of f in the theorem, relying on the boundedness of the variance.

Later, we will see how to do better, by using some ideas closely related to gradient tracking. First, however, let us introduce the batch version of stochastic gradient descent.

6.2.3 Mini-batch and full-batch SGD

There is no reason that prevents us from performing multiple updates of SGD per epoch, especially if we get more data points. In this case, let us introduce the concept of data batch. Assume that at each epoch, you have access to B samples $\theta_i \in \mathcal{B}_i$. Then, the SGD update can be modified upon defining the batch loss,

$$f_{\mathcal{B}_i}(\boldsymbol{x}) = \frac{1}{B}\sum_{i\in\mathcal{B}_i} f_i(\boldsymbol{x}). \tag{6.3}$$

In this way, the batch SGD is the following algorithm.

Batch Stochastic Gradient Descent (Also called SGD)

- *Start with a* $\boldsymbol{x}_0 \in \mathbf{R}^n$, *a learning rate sequence* $\{\alpha_k\}$, *and a finite number of samples* N, *and batches of size* B,
- *Iterate for all epochs* $k = 0, 1, \ldots$: *pick at random batch* i,

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha_k \nabla f_{\mathcal{B}_i}(\boldsymbol{x}_k)$$

This is a small modification and it doesn't change the convergence properties of the algorithm. If the batch size B is less than the total number of points N, then we say we are in the mini-batch setting. In the limit $B = 1$, then we regain the "standard" SGD. For $B = N$, then we are in the full-batch case, and we get back the standard gradient descent, which incurs in zero asymptotical error.

6.2.4 Advanced convergence results for SGD*

We can now prove some more sophisticated convergence results for SGD and its mini-batch version. These theorems and proofs will be instrumental for better understanding of the results in later chapters. They will also show that one has to be extremely careful in the algorithmic setting that yields a particular rate.

Theorem 6.2 (SGD convergence with strong convexity and smoothness) *Consider the problem (EDP) and its optimizer* $\boldsymbol{x}^*$. *Assume* m*-strong convexity and* L*-smoothness of* f. *Assume bounded inter-variance as,*

$$\mathbf{E}[\|\nabla f_i(\boldsymbol{x}) - \frac{1}{N}\sum_{j=1}^{N} \nabla f_j(\boldsymbol{x})\|^2] \leqslant \sigma^2,$$

for a finite constant $\sigma > 0$. *Define the quadratically-weighted average,*

$$\hat{\boldsymbol{x}}_T = \frac{1}{S_T}\sum_{k=0}^{T-1} w_k \boldsymbol{x}_k, \quad \text{for } w_k = (a+k)^2, \quad S_T = \sum_{k=0}^{T-1} w_k \geqslant \frac{1}{3}T^3,$$

Then, for stepsize $\alpha_k = \frac{4}{m(a+k)}$, *with* $a = 8L/m$, *we obtain the following convergence certificate,*

$$\mathbb{E}[f(\hat{\boldsymbol{x}}_T)] - f^* = O\left(\frac{\sigma^2}{mT}\right).$$

Proof. *We follow the proof line of [Sti18], which we adapt to our case. We recall a few results first.*

For L-smooth functions we have,

$$\|\nabla f(\boldsymbol{x}_k) - \nabla f(\boldsymbol{x}^*)\|^2 \leqslant 2L(f(\boldsymbol{x}_k) - f^*). \tag{6.4}$$

This can be shown by letting $\boldsymbol{y} = \boldsymbol{x}^*$ *in Equation* (1.7).

For m-strongly convex functions we have,

$$-\langle \boldsymbol{x} - \boldsymbol{x}^*, \nabla f(\boldsymbol{x}_k)\rangle \leqslant -(f(\boldsymbol{x}) - f^*) - \frac{m}{2}\|\boldsymbol{x} - \boldsymbol{x}^*\|^2 \tag{6.5}$$

which follows from the definition of strong convexity, $f^* \geqslant f(\boldsymbol{x}) + \langle \boldsymbol{x}^* - \boldsymbol{x}, \nabla f(\boldsymbol{x}_k)\rangle + \frac{m}{2}\|\boldsymbol{x} - \boldsymbol{x}^*\|^2$.

For any sequence $\{a_k\}$, $a_k \geqslant 0$, $\{e_k\}$, $e_k \geqslant 0$ *satisfying,*

$$a_{k+1} \leqslant (1 - m\alpha_k)a_k - \alpha_k e_k + \alpha_k^2\sigma^2,$$

for $\alpha_k = \frac{4}{m(a+k)}$, *and constants* $\sigma^2 \geqslant 0$, $m > 0$, $a > 1$, *we have,*

$$\frac{1}{S_T}\sum_{k=0}^{T-1} w_k e_k = O\left(\frac{\sigma^2}{mT}\right), \tag{6.6}$$

with the same definitions of S_T and w_k in the statement of the theorem. The proof of which can be found in [SCJ18].

Now, define $\boldsymbol{g}_k = \nabla f_i(\boldsymbol{x}_k)$, and $\bar{\boldsymbol{g}}_k = \frac{1}{N}\sum_{j=1}^N \nabla f_j(\boldsymbol{x}_k)$. Define also the loss to minimize as $f(\boldsymbol{x}) = \frac{1}{N}\sum_{j=1}^N \nabla f_j(\boldsymbol{x})$, and consequently $\bar{\boldsymbol{g}}_k = \nabla f(\boldsymbol{x}_k)$.

From the update rule,

$$\begin{aligned}\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2 &= \|\boldsymbol{x}_k - \alpha_k \boldsymbol{g}_k - \boldsymbol{x}^*\|^2 \\ &= \|\boldsymbol{x}_k - \alpha_k \bar{\boldsymbol{g}}_k - \boldsymbol{x}^*\|^2 + \alpha_k^2\|\boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\|^2 - 2\alpha_k\langle \boldsymbol{x}_k - \alpha_k\bar{\boldsymbol{g}}_k - \boldsymbol{x}^*, \boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\rangle.\end{aligned}$$

Observe now that,

$$\begin{aligned}\|\boldsymbol{x}_k - \alpha_k\bar{\boldsymbol{g}}_k - \boldsymbol{x}^*\|^2 &= \|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 + \alpha_k^2\|\bar{\boldsymbol{g}}_k\|^2 - 2\alpha_k\langle \boldsymbol{x}_k - \boldsymbol{x}^*, \bar{\boldsymbol{g}}_k\rangle \\ &= \|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 + \alpha_k^2\|\bar{\boldsymbol{g}}_k - \nabla f(\boldsymbol{x}^*)\|^2 - 2\alpha_k\langle \boldsymbol{x}_k - \boldsymbol{x}^*, \bar{\boldsymbol{g}}_k\rangle.\end{aligned}$$

We use now (6.4) and (6.5) to say,

$$\begin{aligned}\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2 &\leqslant \|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2 + \alpha_k^2\|\boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\|^2 + 2\alpha_k^2 L(f(\boldsymbol{x}_k) - f^*) - 2\alpha_k(f(\boldsymbol{x}_k) - f^*) \\ &\qquad -\alpha_k m\|\boldsymbol{x}^k - \boldsymbol{x}^*\|^2 - 2\alpha_k\langle \boldsymbol{x}_k - \alpha_k\bar{\boldsymbol{g}}_k - \boldsymbol{x}^*, \boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\rangle \\ &\leqslant (1-\alpha_k m)\,\|\boldsymbol{x}^k - \boldsymbol{x}^*\|^2 + 2\alpha_k(f(\boldsymbol{x}_k) - f^*)(\alpha L - 1) + \alpha_k^2\|\boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\|^2 \\ &\qquad -2\alpha_k\langle \boldsymbol{x}_k - \alpha_k\bar{\boldsymbol{g}}_k - \boldsymbol{x}^*, \boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\rangle.\end{aligned}$$

For $\alpha_k \leqslant \frac{1}{2L}$, then, $\alpha L - 1 \leqslant -\frac{1}{2}$, therefore taking expectations ($\mathbb{E}[\boldsymbol{g}_k - \bar{\boldsymbol{g}}_k\|^2] \leqslant \sigma^2$, $\mathbb{E}[\boldsymbol{g}_k] = \bar{\boldsymbol{g}}_k$),

$$\mathbb{E}[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] \leqslant (1-\alpha_k m)\,\mathbb{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] - \alpha_k\mathbb{E}[f(\boldsymbol{x}_k) - f^*] + \alpha_k^2\sigma^2 \tag{6.7}$$

Use now the special supporting sequence (6.6) with $a = 8L/m$, so that $\alpha_k \leqslant 1/2L$, and the theorem is proven. ♣

The theorem is very interesting for a few reasons. First, it establishes a $O(1/T)$ convergence for strongly convex and smooth functions. This is particularly useful for us. Note that in this case, we do not get back the linear convergence that one could expect in deterministic cases. This also means that better algorithms are possible, as we will see shortly. Second, it uses a cleverly constructed quadratic sequence of weights. This is rather important: traditionally, one would use a linear sequence of weight (a so-called ergodic mean). Using quadratic (or polynomially)-dependent sequence can offer better guarantees, sometimes.

The next theorem we present shows the best achievable error with convexity and smoothness alone. This best error, as often in machine learning, is at fixed time T_{f}. That is, fixing the number of iterations T_{f}, what is the best error you can achieve?

Theorem 6.3 (SGD convergence with smoothness alone, fixing $T = T_{\mathrm{f}}$) *Consider the problem (EDP) and its optimizer $\boldsymbol{x}^*$. Assume convexity and L-smoothness of f. Assume bounded inter-variance as,*

$$\mathbf{E}[\|\nabla f_i(\boldsymbol{x}) - \frac{1}{N}\sum_{j=1}^N \nabla f_j(\boldsymbol{x})\|^2] \leqslant \sigma^2,$$

for a finite constant $\sigma > 0$. Fix the time horizon $T = T_{\mathrm{f}}$. Define the weighted average,

$$\hat{\boldsymbol{x}}_T = \frac{1}{T}\sum_{k=0}^{T-1} \boldsymbol{x}_k\,.$$

Then, the best error is achieved for constant stepsize $\alpha = \min\{\frac{1}{2L}, \frac{B}{\sigma\sqrt{T_{\mathrm{f}}}}\}$, with $B^2 = \mathbb{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2]$, and we obtain the following convergence certificate,

$$\mathbb{E}[f(\hat{\boldsymbol{x}}_{T_{\mathrm{f}}})] - f^* = O\left(\frac{L}{T_{\mathrm{f}}}\right) + O\left(\frac{\sigma}{\sqrt{T_{\mathrm{f}}}}\right) = O\left(\frac{1}{\sqrt{T_{\mathrm{f}}}}\right).$$

Proof. *We can use the same proof of the previous theorem, in particular Eq. (6.7) to arrive at ($m = 0$),*

$$\mathbb{E}[\|\boldsymbol{x}_{k+1} - \boldsymbol{x}^*\|^2] \leqslant \mathbb{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] - \alpha_k \mathbb{E}[f(\boldsymbol{x}_k) - f^*] + \alpha_k^2 \sigma^2,$$

and summing over k,

$$\sum_{k=0}^{T-1} \alpha_k \mathbb{E}[f(\boldsymbol{x}_k) - f^*] \leqslant \mathbb{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2] + \sum_{k=0}^{T-1} \alpha_k^2 \sigma^2.$$

With this and Jensen's inequality, we write,

$$\mathbb{E}[f(\hat{\boldsymbol{x}}_T) - f^*] \leqslant \frac{\sum_{k=0}^{T-1} \alpha_k \mathbb{E}[f(\boldsymbol{x}_k) - f^*]}{\sum_{k=0}^{T-1} \alpha_k} \leqslant \frac{\mathbb{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2]}{\sum_{k=0}^{T-1} \alpha_k} + \frac{\sum_{k=0}^{T-1} \alpha_k^2 \sigma^2}{\sum_{k=0}^{T-1} \alpha_k},$$

with $\hat{\boldsymbol{x}}_T = \frac{\sum_{k=0}^{T-1} \alpha_k \boldsymbol{x}_k}{\sum_{k=0}^{T-1} \alpha_k}$. Fixing $T = T_\mathrm{f}$, we can optimize the right-hand side (the error). This is a convex optimization problem in $\alpha \in (0, 1/(2L)]$ [BXM03], with solution $\alpha_k = \alpha = \min\{\frac{1}{2L}, \frac{B}{\sigma\sqrt{T_\mathrm{f}}}\}$, and $B^2 = \mathbb{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2]$. The best error is then,

$$\begin{aligned}
\frac{\mathbb{E}[\|\boldsymbol{x}_0 - \boldsymbol{x}^*\|^2]}{\sum_{k=0}^{T-1} \alpha_k} + \frac{\sum_{k=0}^{T-1} \alpha_k^2 \sigma^2}{\sum_{k=0}^{T-1} \alpha_k} &= \frac{B^2}{T_\mathrm{f}\alpha} + \alpha\sigma^2 \\
&= \frac{B^2}{T_\mathrm{f}} \max\{2L, \frac{\sigma\sqrt{T_\mathrm{f}}}{B}\} + \sigma^2 \min\{\frac{1}{2L}, \frac{B}{\sigma\sqrt{T_\mathrm{f}}}\} \\
&\leqslant \frac{2LB^2}{T_\mathrm{f}} + 2B\frac{\sigma}{\sqrt{T_\mathrm{f}}}
\end{aligned}$$

as we wanted to prove. ♣

The theorem tells you the best error you can achieve fixing the horizon and optimizing the stepsize accordingly. It gives you a good idea of what is achievable at best. If you were able to derive results so that the stepsize didn't depend on T_f, then the results would be valid for any T_f and therefore asymptotically[1].

Sometimes, in machine learning, we say that the rate of SGD for smooth convex function is $O(L/T) + O(\sigma/\sqrt{T})$. This has to be intended as best achievable error rate, which (for the moment) remains a theoretical bound, not achievable by the SGD we have presented.

Sometimes, we also say that the first term $O(L/T)$ is the bias, and the second term $O(\sigma/\sqrt{T})$ is the error variance.

By using Nesterov's acceleration for smooth functions in SGD to design an accelerated-SGD, we can also obtain,

$$\mathbb{E}[f(\hat{\boldsymbol{x}}_{T_\mathrm{f}})] - f^* = O\left(\frac{L}{T_\mathrm{f}^2}\right) + O\left(\frac{\sigma}{\sqrt{T_\mathrm{f}}}\right),$$

as one could expect. These results are good lower bound on what is achievable and can give you good rules of thumbs to design your algorithms. If you are interested to know more, please refer to [DGBSX12, GL13] and subsequent publications.

Finally, let us reconsider the SGD convergence in Theorem 6.1 for a fixed time T_f. By choosing $\alpha = \frac{1}{2mT_\mathrm{f}}$, one obtains an error ball of size $O(1/T_\mathrm{f})$. We see therefore that the error $O(1/T)$ is a theoretical lower bound, while we can achieve a rate of $O(\log(T)/T)$ with $\alpha = \frac{1}{2mk}$.

6.3 Variance reduction and the SAGA algorithm

One of the issues we have with SGD is that it uses directly the gradient $\nabla f_i(\boldsymbol{x})$ as approximator of the total gradient. This is similar to DGD when it was using the local gradients as approximators of the total gradient. This approach delivers a $O(1/T)$ rate for smooth strongly convex functions,

[1]Sometimes, machine learning researchers use a clever re-initialization strategy, the so-called doubling-trick, to render some fixed-time results any-time results.

which we know it is not the optimal rate for this class. In fact, we would expect a linear convergence in this case as optimal. As we saw for gradient tracking, a way out is to use another proxy: use $\nabla f_i(\boldsymbol{x})$ to update an estimate $\boldsymbol{g} \approx \nabla f(\boldsymbol{x}) = \frac{1}{N}\sum_{i=1}^N \nabla f_i(\boldsymbol{x})$. We see that this strategy pays out and delivers the optimal rate.

6.3.1 Algorithm

Let us introduce the algorithm. Start with a $\boldsymbol{x}_0 \in \mathbf{R}^n$ and a constant learning rate α. Initialize the gradient estimate as, $\boldsymbol{g}_0 = \frac{1}{N}\sum_{i=1}^N \nabla f_i(\boldsymbol{x}_0)$.

SAGA Algorithm

- *Start with a $\boldsymbol{x}_0 \in \mathbf{R}^n$, a constant learning rate sequence α, and a finite number of samples N. Initialize the gradient estimate as, $\boldsymbol{g}_0 = \frac{1}{N}\sum_{i=1}^N \nabla f_i(\boldsymbol{x}_0)$.*
- *Iterate for all epochs $k = 1, \ldots$:*
  - *Pick at random sample i,*
  - *Update:*
    $$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \boldsymbol{v}, \qquad \textit{with } \boldsymbol{v} = \boldsymbol{g}_{k-1} + [\nabla f_i(\boldsymbol{x}_k) - \nabla f_i(\boldsymbol{x}_{k-1})]$$
  - *Update:*
    $$\boldsymbol{g}_k = \boldsymbol{g}_{k-1} + \frac{1}{N}[\nabla f_i(\boldsymbol{x}_k) - \nabla f_i(\boldsymbol{x}_{k-1})]$$

We see how in the algorithm we update the proxy $\boldsymbol{g}_k$ with the gradient of the sample that is picked. We see also the appearance of a $1/N$ factor for the update of $\boldsymbol{g}_k$ but not in the update $\boldsymbol{v}$. The reason is technical and it delivers an unbiased update. Adding $1/N$ into the update of $\boldsymbol{v}$ delivers a biased algorithm (SAG, which enjoys less variance). See discussion in [DBLJ14].

6.3.2 Convergence guarantees

We have now the following theorem.

Theorem 6.4 (SAGA convergence) *Consider the problem (EDP) and its optimizer $\boldsymbol{x}^*$. Assume L-smoothness of f with respect to $\boldsymbol{x}$. Define the ergodic sequence, $\hat{\boldsymbol{x}}_k = \frac{1}{k}\sum_{t=1}^k \boldsymbol{x}_t$. Then, SAGA with a small enough constant learning rate $\alpha \leqslant 1/(3L)$ will generate a sequence $\{\boldsymbol{x}_k\}$ which converges as,*

$$\mathbf{E}[f(\hat{\boldsymbol{x}}_k) - f^*] \leqslant O\left(\frac{1}{k}\right).$$

Assume now m-strong convexity and L-smoothness of f with respect to $\boldsymbol{x}$.

Then, SAGA with a small enough constant learning rate $\alpha \leqslant 1/(3L)$ will generate a sequence $\{\boldsymbol{x}_k\}$ which converges as,

$$\mathbf{E}[\|\boldsymbol{x}_k - \boldsymbol{x}^*\|^2] \leqslant O(\rho^k),$$

with $\rho \in (0, 1)$.

Proof. *See [DBLJ14].* ♣

The theorem says that we can reach zero error in a linear converging fashion by using SAGA on strongly convex smooth functions (at the price of maintaining another vector $\boldsymbol{g}_k$ in memory). This is very similar to Gradient Tracking. In this respect, SAGA reaches the optimal rate in strongly convex smooth problems, so you can't really improve on it. However, for non-convex many other methods exist (for instance, you will encounter Adam: adaptive momentum estimator).

SAGA also reaches $O(1/k)$ for smooth functions, which is the optimal rate for non-accelerated algorithms, thereby beating the $O(1/\sqrt{k})$ for SGD.

We will see how SGD and SAGA will play a crucial role in cooperative learning next.

6.4 Numerical example: Kernel regression

As usual, we come back to the kernel regression example and we implement SGD and SAGA. Here, I consider each devices to be selected randomly (the i) and each of them has its own number of points. This is the so-called mini-batch setting: N does not represent the number of data points, but N batches of data points.

I select the learning rate as $s = 0.002$, and I plot everything in Figure 6.1.

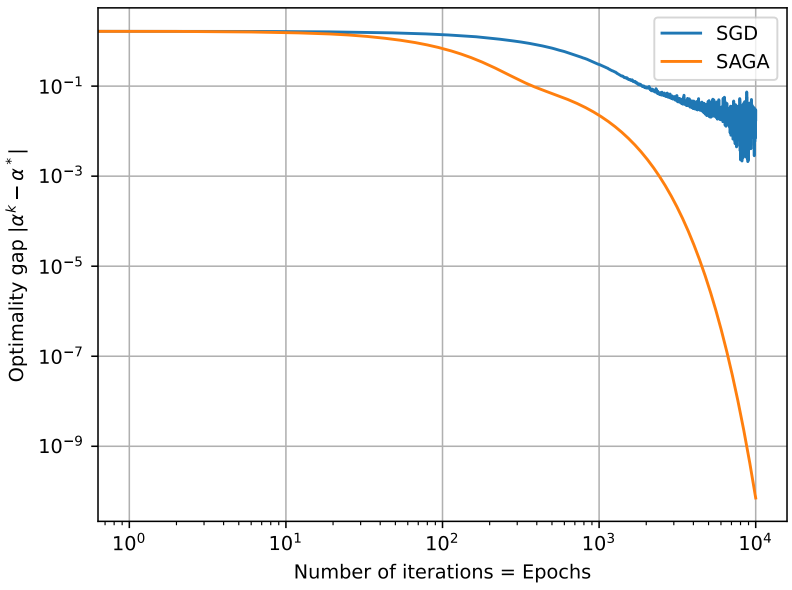


Figure 6.1. *Convergence of SGD and SAGA with a constant learning rate of* $s = 0.002$ *on a mini-batch version of the Kernel regression problem.*

Chapter 7

Federated learning: the basics

7.1 Introduction

In this chapter, we are going to study one of the newest cooperative learning technique: federated learning (FL). Under some angle, FL is the machine learning variant of distributed optimization, and yet it is very different in spirit. First of all, it uses distributed optimization, but a version of it that is brought to the extreme, with very large-scale datasets, almost no communication, and very simple global computations.

FL was proposed in 2016/2017 by Google researchers working with academia, and it is now a very vibrant research endeavour with thousands of papers a year. It is one the hottest research areas in machine learning. Here, the aim is to give you a sense of what FL is, its basic components and algorithms, and pointers to some more advanced topics. We will be following the survey [KM$^+$21], and I will specify some other papers as we go along.

7.1.1 The setting revisited

We start by reconsidering the risk minimization problem,

$$\text{(EP)} \qquad \min_{\boldsymbol{x}\in X\subseteq\mathbf{R}^n} \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)],$$

which we have studied in Chapter 6. We recall that f is here a convex function in $\boldsymbol{x}$, parametrized by a random vector θ, of distribution Θ. The convexity of f in $\boldsymbol{x}$ makes the problem convex and the setting safe for us to analyze mathematically. We note already that FL extends well beyond this safe setting into the non-convex domain, but there theoretical guarantees are hard to derive and we will only report some empirical numerical considerations.

We also recall here the sampled problem, i.e., the empirical risk minimization problem with N samples, as

$$\text{(EPS)} \qquad \min_{\boldsymbol{x}\in X\subseteq\mathbf{R}^n} \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)] \approx \frac{1}{N}\sum_{i=1}^{N} f_i(\boldsymbol{x}),$$

where, $f_i(\boldsymbol{x}) := f(\boldsymbol{x};\theta_i)$. This latter problem is the only one we can solve, and it is the one we will be focusing on mainly.

The idea behind federated learning is to consider a number of devices (ofter referred to as workers or clients) that train local models and communicate their model weights to a cloud service (or central coordinator, or server). It is useful to redefine some "common" words.

Definition 7.1 (Model) *In machine learning, we say that somebody trains a model, if they fit it to data. Imagine you have some features $x \in \mathbf{R}^n$ and labels $y \in \mathbf{R}^m$. You can design a model, which is an hypothetical relationship between features and labels (e.g., a parametric function):*

$$y \approx F(x;\boldsymbol{w}).$$

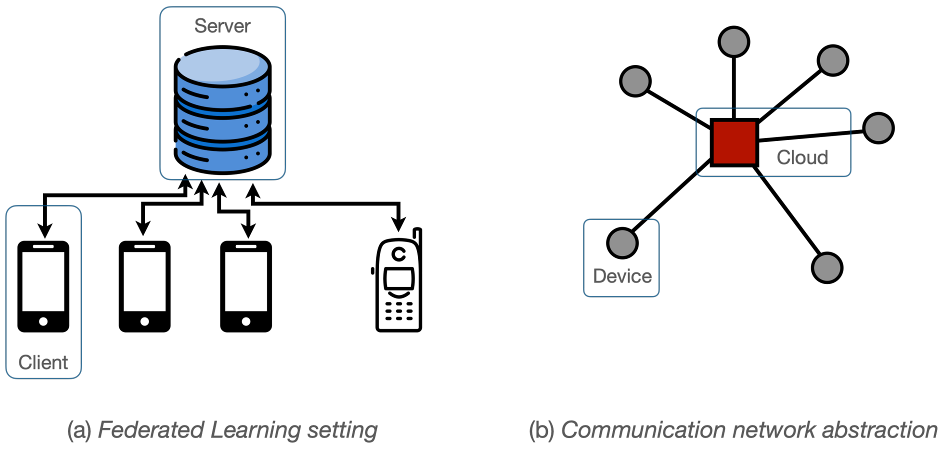

Server
Client
Cloud
Device
(a) *Federated Learning setting*
(b) *Communication network abstraction*

Figure 7.1. *A pictorial depiction of the federated learning setting and its cloud-device representation.*

The model is parametrized by weights $\boldsymbol{w} \in \mathbf{R}^p$. The models are constructed based on experience and common architectures. Linear models are the easiest: $y \approx Ax + b$, where (A, b) can be packed into the model weight vector $\boldsymbol{w}$. Deep neural networks are the most complex models.

Training a model is the optimization problem,

$$\underset{\boldsymbol{w} \in \mathbf{R}^p}{\text{minimize}}\, \ell(x, y; \boldsymbol{w}),$$

where ℓ is a loss function. For example, it can be the least squares loss: $\ell(x, y; \boldsymbol{w}) = \frac{1}{2}\|y - F(x; \boldsymbol{w})\|_2^2$.

When we say the clients share their model weights, or their weights, or loosely their model, we always refer to them sharing their vector $\boldsymbol{w}$ and each of the model function F being the same across clients.

Let us look at Figure 7.1. On the left, you can see the simplified FL setting, where a number of clients, possibly heterogenous, collect data and send their model weights to a centralized server who processes them. On the right, you can see an equivalent representation in the form of a cooperative problem perspective, with devices and the cloud. This is the baseline setting, then we will see that peer-to-peer variants start to exist.

7.1.2 Problem statement and main messages

Let us try to formalize the cooperative problem at hand mathematically. Look back at the risk minimization problem, but now distributed among C clients,

$$\begin{aligned}
\underset{\boldsymbol{x} \in X \subseteq \mathbf{R}^n}{\text{minimize}} \quad & \mathbf{E}_{\theta \sim \Theta}[f(\boldsymbol{x}; \theta)] \\
& \equiv \frac{1}{C}\sum_{c=1}^{C} \mathbf{E}_{\theta \sim \Theta}[f(\boldsymbol{x}; \theta)] && (7.1) \\
& \approx \frac{1}{C}\sum_{p=1}^{C} \mathbf{E}_{\theta \sim \Theta_c}[f(\boldsymbol{x}; \theta)] && (7.2) \\
& \approx \frac{1}{C}\sum_{p=1}^{C} \frac{1}{N_c}\sum_{j=1}^{N_c} f_j(\boldsymbol{x}) = \frac{1}{C}\sum_{c=1}^{C} f_c(\boldsymbol{x}). && (7.3)
\end{aligned}$$

A few comments are now in order to explain the derived equations. The first step (7.1) is exact: we are distributing the risk among the C clients. The second step (7.2) is the first real federated learning step: we are relaxing the assumption that the data θ is generated by the same distribution and we allow for heterogenous devices. The samples θ coming from different devices are therefore not IID (independent and identically distributed).

The third step (7.3) is the second federated learning step: we are relaxing the assumption that the training size (i.e., the number of data points) is the same across clients.

The final result of these two relaxations looks like our cooperation problem: it is in form, but hardly in essence. I will keep the C instead of N for federated learning problems to mark a stark difference between the two problems. While,

$$\underset{\boldsymbol{x}\in X\subseteq \mathbf{R}^n}{\text{minimize}}\ \frac{1}{C}\sum_{c=1}^{C} f_c(\boldsymbol{x}), \tag{7.4}$$

can be solved with a distributed optimization algorithm, the importance here is also how the functions f_c were generated and whether it reasonable to solve it with respect to the starting problem we wanted to solve, which had the training loss $\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)]$.

This is the first main message: federated learning is less interested in solving the cooperative problem (7.4) exactly than finding ways to solve the risk minimization problem,

$$(\mathsf{EP}) \qquad \underset{\boldsymbol{x}\in X\subseteq \mathbf{R}^n}{\text{minimize}}\ \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)],$$

by using (7.4) as a proxy. In this context, it makes sense to ask questions related to data heterogeneity. For example think taking two pictures with an old phone, and a thousands with the newest phone you can find. Are the models obtained by the two devices comparable?

The second main message is then the following. Since solving (EP) is often unattainable, federated learning is more interested in obtaining good local models that are refined via collaborating with other devices. This is the concept of a federation. Every devices "tries to do the best it can", and "sometimes" it collaborates with the cloud to refine and improve its model.

7.2 FedAvg: the baseline algorithm of FL

We are now in shape to introduce our first federated learning algorithm. The algorithm is a parallel (or Jacobi-type) algorithm that merges the model weights coming from the different devices by simple averaging. The algorithm tackles Problem (7.4), in a cloud-based setting. First, let's introduce some new nomenclature proper to FL, which we collect in the table below.

Optimization		Federated Learning
Devices	$\Longleftrightarrow$	Clients
Cloud/central node	$\Longleftrightarrow$	Server
Stepsize	$\Longleftrightarrow$	Learning rate
		Batch size B: number of data points each client uses to update the local model
		Local epochs E: times local data is used to update the local models

The algorithm is the following.

Federated averaging (FedAvg)

Server update.

Initialize $\boldsymbol{x}_0$. For each time $t = 1, \ldots$ do:

- *Select $P \leqslant C$ clients, and for each client*
  - **Communication to client step:** *Send $\boldsymbol{x}_t$ to client*
  - **Communication from client step:** *Receive $\boldsymbol{x}^i_{t+1}$ from* **client update**
- **Computation step:** *Perform the mixing*

$$\boldsymbol{x}_{t+1} = \sum_{p=1}^{P} \frac{N_p}{\sum_{p=1}^{P} N_p} \boldsymbol{x}^p_{t+1}$$

Client update.

Start from $\boldsymbol{x}^i_0 = \boldsymbol{x}_t$. Select batches $\mathcal{B}$ of data, each containing B data points.

For epoch $k = 1, \ldots, E$

- *Select a data batch $\mathcal{B}_j$ and perform a local SGD update as,*

$$\boldsymbol{x}^i_{k+1} = \boldsymbol{x}^i_k - \alpha_k \nabla f_{\mathcal{B}_j}(\boldsymbol{x}^i_k),$$

Several explanations are now in order. In the algorithm, we have two sides: a server side and a client side. We have also two timescale, the server counter t, and the client counter k. The former can be unbounded, since we can run the algorithm indefinitely. The latter is bounded by the number of epochs E. The clients run a local SGD based on the local data, and initialized by the server model weights. We recall the mini-batch setting and loss (6.3). The server averages the weights based on the number of samples each device has (N_p). This last adjustment is to account for data unbalancedness.

The local SGD are run on data samples of batch size B. In par with traditional SGD, B can be one, and we use only one data sample per iteration.

FedAvg is a Jacobi algorithm with a very simple update rule. It has many tuning knobs and in practice clients can come in and drop out randomly, and there is no need to have complicated optimization problems in the server. Of course, we don't expect great convergence rates from it! But, that's hardly the point in FL.

FedAvg can be rewritten in the matrix-vector form as follows. Stack the model weights into the vector $\boldsymbol{y}$, and the t-th mixing into the doubly-stochastic matrix $\mathbf{W}_t$. Define the function $F(\boldsymbol{y}) = \frac{1}{C}\sum_{c=1}^{C} f_c(\boldsymbol{x}^c)$, and $F_k(\boldsymbol{y})$ which is a batch version of it, with a randomly selected batch of data.

Then FedAvg reads,

$$\boldsymbol{y}_{t+1} = \mathbf{W}_t \left[\prod_{k=1}^{E} (I - \alpha_k \nabla F_k)\right] \circ \boldsymbol{y}_t, \tag{7.5}$$

where we have used an operation theoretic notation, to say that we apply E local SGD before a mixing step.

Written as it is, we can see that FedAvg is distributed gradient descent (see Chapter 2.2.5) on the timescale t, and local SGD on the timescale k. In particular, the distributed gradient descent is edge asynchronous, since $\mathbf{W}_t$ can change at each t, since different clients are selected.

So, FedAvg can fall back to two of the algorithms we have already seen (it is a generalization thereof). Distributed gradient descent with constant $\mathbf{W}$ and single epochs E, which amounts at implementing batch SGD on the whole dataset (all the devices), and Distributed gradient descent with constant $\mathbf{W}$ and single epochs E but with full batch, which amounts at implementing a standard gradient descent. The latter achieves zero error linearly for $f \in \mathcal{S}^{1,1}_{m,L}(\mathbf{R}^n)$.

FedAvg has many control knobs and therefore there are many variants in the literature. For instance, we may tune the amount of communication in $\mathbf{W}_t$ in a non-trivial way (e.g., increasing it or decreasing it per iteration), we may use time-dependent batch sizes, time-dependent epochs, and so on, and so forth.

7.2.1 FedAvg: convergence

Let us analyze now the convergence of FedAvg. In general, this research area is very active, and we only present two sample results.

First, let us put ourselves in the data IID setting, so that we can define the loss

$$F(\boldsymbol{x}) := \frac{1}{C}\sum_{c=1}^{C}\mathbf{E}_c[f(\boldsymbol{x};\theta)] = \mathbf{E}[f(\boldsymbol{x};\theta)].$$

In this setting, FedAvg (sometimes referred to as parallel SGD or local SGD) competes with two natural baselines. First, we may keep $\boldsymbol{x}$ fixed in local updates during each round, and compute a total of CE gradients at the current $\boldsymbol{x}$ (E per each clients), in order to run accelerated minibatch SGD. We then have the upper bound

$$\mathbb{E}[F(\hat{\boldsymbol{x}}_T)] - F^* < O\left(\frac{L}{T^2}\right) + O\left(\frac{\sigma}{\sqrt{TCE}}\right),$$

for the average of the iterated $\hat{\boldsymbol{x}}_T$ and smooth convex objectives (see Chapter 6.2.4).

A second natural baseline is to ignore all but 1 of the C active clients, which allows (accelerated) sequential SGD to execute for ET steps. Applying the same general bounds cited above, this approach offers an upper bound of

$$\mathbb{E}[F(\hat{\boldsymbol{x}}_T)] - F^* < O\left(\frac{L}{(ET)^2}\right) + O\left(\frac{\sigma}{\sqrt{ET}}\right).$$

Comparing these two results, we see that minibatch SGD attains the optimal "statistical" term $O\left(\frac{\sigma}{\sqrt{TCE}}\right)$, whilst SGD on a single device (ignoring the updates of the other devices) achieves the optimal "optimization" term $O\left(\frac{L}{(ET)^2}\right)$.

For FedAvg, we have the following theorem.

Theorem 7.1 (FedAvg convergence (special case I)) *Consider using FedAvg to solve the problem* $\min_{\boldsymbol{x}\in\mathbf{R}^n} F(\boldsymbol{x})$ *with* C *clients equipped with IID data. Let the local* f *be convex and* L*-smooth. Assume bounded variance as,*

$$\mathbf{E}[\|\nabla f(\boldsymbol{x};\theta) - \nabla F(\boldsymbol{x})\|^2] \leqslant \sigma^2.$$

Assume also local gradient boundedness

$$\mathbf{E}[\|\nabla f_c(\boldsymbol{x}) - \frac{1}{N}\sum_{c=1}^{C}\nabla f_c(\boldsymbol{x}^*)\|^2] = \mathbf{E}[\|\nabla f_c(\boldsymbol{x})\|^2] \leqslant G^2,$$

for a finite constant $G > 0$*, as done in Theorem 6.1.*

Then, FedAvg can be made converge at fixed time as,

$$\mathbf{E}[F(\hat{\boldsymbol{x}}_T) - F(\boldsymbol{x}^*)] \leqslant \underbrace{O\left(\frac{LCE}{T_{\mathrm{f}}}\right)\frac{G^2}{\sigma^2}}_{bias} + \underbrace{O\left(\frac{\sigma}{\sqrt{CET_{\mathrm{f}}}}\right)}_{variance} < O(1/\sqrt{TCE}),$$

where T *is the number of outer iterations and* $\hat{\boldsymbol{x}}_T$ *is the standard ergodic mean of the iterates.*

If we further assume that the local f *are strongly convex, then a suitable average sequence* $\bar{\boldsymbol{x}}_k$ *can be made converge as*

$$\mathbf{E}[F(\bar{\boldsymbol{x}}_T) - F(\boldsymbol{x}^*)] = O\left(\frac{\sigma^2}{mTCE}\right),$$

with a decreasing learning rate $\alpha_k = O(1/tk)$*.*

Proof. *The proofs are generalizations of the proofs of Theorems 6.2-6.3. See also [Sti18] and [KM$^+$21]. Be careful that sometimes researchers define T as the total number of gradient evaluations, and not the outer counter as we do here.* ♣

The theorem states that we can make FedAvg converge to zero error, but it can be slow in a non-strongly convex setting. Let us now consider next the non-IDD data setting.

Theorem 7.2 (FedAvg convergence (special case II)) *Consider using FedAvg to solve the problem*

$$\underset{\boldsymbol{x}\in\mathbf{R}^n}{\text{minimize}}\, F(\boldsymbol{x}) := \frac{1}{C}\sum_{c=1}^{C}\mathbf{E}_c[f(\boldsymbol{x};\theta)] \neq \mathbf{E}[f(\boldsymbol{x};\theta)],$$

with C clients equipped with non-IID data. Let the local f be strongly convex and L-smooth. Assume bounded variance as,

$$\mathbf{E}[\|\nabla f(\boldsymbol{x};\theta) - \nabla F(\boldsymbol{x})\|^2] \leqslant \sigma^2,$$

and bounded square gradient as $\mathbf{E}[\|\nabla f(\boldsymbol{x};\theta)\|^2\| \leqslant G^2$. Then, upon defining a suitable learning rate sequence $\alpha_k = O(1/tk)$, FedAvg can be made converge as,

$$\mathbf{E}[F(\boldsymbol{x}_k) - F(\boldsymbol{x}^*)] \leqslant O(1/T).$$

Furthermore, to attain an accuracy ϵ, then the number of required communications T are,

$$T = \frac{1}{\epsilon}O\left(EG^2 + \frac{\sigma^2 + \mathit{\Gamma} + G^2}{E} + G^2\right),$$

where $\mathit{\Gamma} > 0$ is a constant measuring the degree of non-IIDness.

Proof. *See [LHY$^+$20]. Once again, be careful on how T is defined.* ♣

The theorem says that we can still attain convergence in the non-IID setting, but now we are converging to a different loss function than the one we are really interested. The result is similar to the strong convex case of the IID setting, even though the difference is hidden in the $O(\cdot)$ term. In the non-IID case, we have a $\mathit{\Gamma}$ term that measures how different are the distributions from different clients. This term induces an interesting behavior on the communication rounds to obtain a certain accuracy. In fact, the trade-off goes as,

$$T \sim O\left(\frac{1}{E}\right) + O(E),$$

which tells us about a trade-off between local computations and how many mixing steps one should have. To reduce communication, we need to find the best number of local epochs E. Fixing $E = 1$ or $E \gg 1$ may not be the best choice. This issue is not present in such a way in the IID setting, where each client are working on the same data distribution. There, the more E, it was helping global convergence, in the non-IID case, it is not so.

While we have presented here some sample results, there is a lot of research in getting better constants, as well as variance reduction via memory, and so forth. But at the end, I don't think convergence is something we care deeply in federated learning, especially in realistic settings.

7.3 Numerical examples

7.3.1 Kernel setting

Let us revisit our kernel setting applying FedAvg to it. In Figure 7.2, we report the full batch SGD case, where at each time t we consider the updates of all the clients (i.e., complete participation). Here, for $E = 1$ we regain distributed gradient descent with full averaging $\mathbf{W}$, so the error goes to zero asymptotically, the others plateau. Notice that $B = 20$ is the full data set for each devices, so we are in the full batch case.

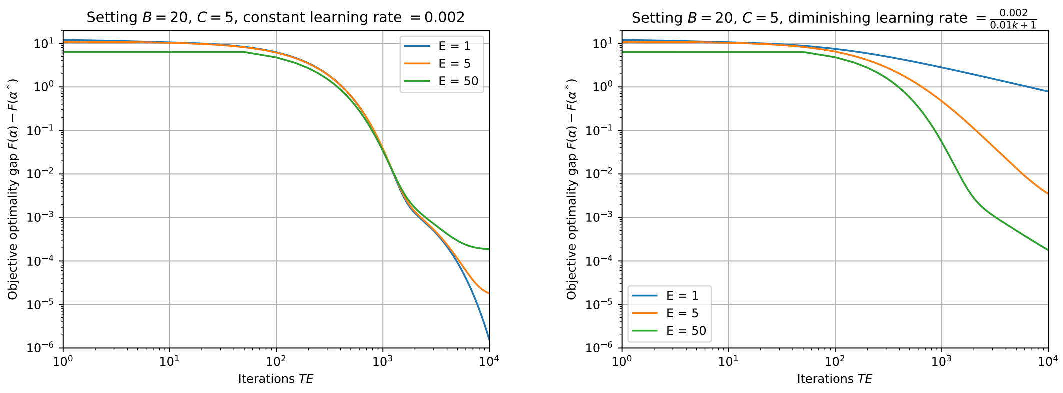


Figure 7.2. *Convergence of FedAvg in the full batch, complete participation $C = 5$ setting.*

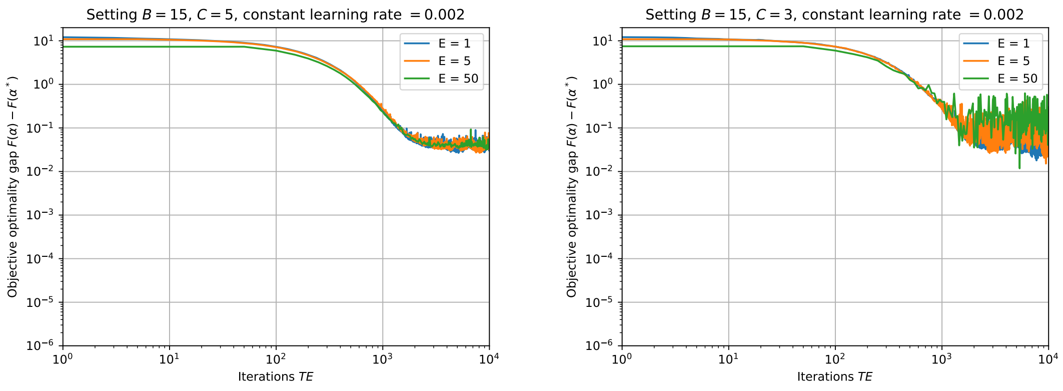


Figure 7.3. *Convergence of FedAvg in the mini batch, partial participation settings.*

Since we are in the strongly convex case, when we define a diminishing learning rate, then the error goes to zero as $O(1/T)$ but it could be slow.

In Figure 7.3, we look at two more realistic assumptions. On the left, we plot the error for the mini-batch SGD case ($B = 15 < 20$). On the right, we consider partial participation, by asking only $C = 3$ clients per step (the total number of clients is 5). On the left, for $E = 1$, we get back mini-batch SGD.

While these figures seem to indicate that FedAvg performs poorly, it is interesting to look at the number of communication rounds to obtain a certain accuracy ϵ, which we do in Figure 7.4. Remember that in FL we are more interested in how many communication rounds do I need to perform to get a better model than if I was training it alone. In this context, Figure 7.4 is more important than any other convergence plot. There, we can see that, if we fix say $CT \sim 100$ (which could be our time budget), FedAvg with partial communication and mini batch is better than distributed gradient descent.

So Figure 7.4 depicts a rather nuanced picture of FedAvg, whose tuning is highly dependent of our particular setting and application.

7.3.2 MNSIT setting

We report in Figure 7.5 the results obtained by [LHY+20] with FedAvg on the MNSIT database with the cross-entropy loss, shared among different clients. In the figure, they indicate with K the number of clients, and their T/E correspond to our T. We also see how they indicate the global (training) loss as a metric, and not the optimality gap. Often this is the case in machine learning, where the actual F^* is not possible to compute, and it is less important than doing better than when you started. For the interested reader their code is available on GitHub `https://github.com/lx10077/fedavgpy/blob/master/README.md`, which also contains exam-

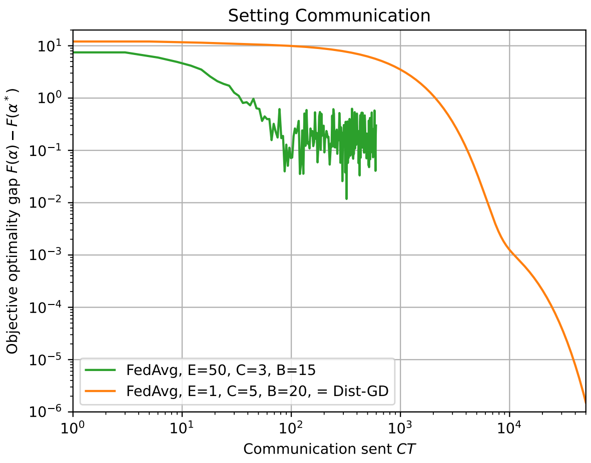


Figure 7.4. *Convergence of FedAvg in terms of communication rounds.*

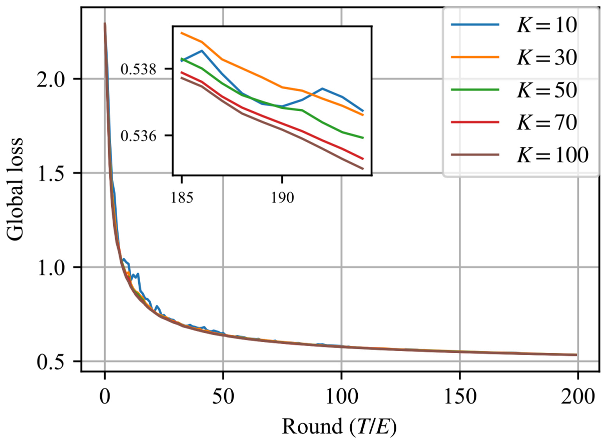


Figure 7.5. *Results obtained by [LHY+ 20] with FedAvg on the MNSIT database with the cross-entropy loss. Their K is the number of clients, and their T/E correspond to our T*

ples with non-convex losses and architectures, such as CNNs.

7.4 SCAFFOLD

7.4.1 Federated Learning with non-IID data

Let us move now to the full real of non IID data, so in the case in which the devices are heterogeneous and collect data that is sensibly different. We have seen how FedAvg can “converge” even in this setting. But we shouldn’t be really happy about the results. Convergence goes as $O(1/T)$ in the strongly convex case, which is not the optimal rate for this class of problems, and also convergence is to a problem that it is not our main focus.

Before deriving the algorithm, let us introduce two settings that are important in federated learning.

Definition 7.2 (Cross-device Federated learning) *We define Cross-device Federated learning the setting in which we have a huge number of small devices (like phones) in an IoT setting. If all the devices are identical, then you could assume IID data. If the devices are very different then you cannot assume IID data. The IID assumption here can save communication rounds and complexity.*

Definition 7.3 (Cross-silo Federated learning) *We define Cross-silo Federated learning the* *setting in which we have very few agents who carry a huge quantity of data. Think of it as FL between companies or hospitals. Each of the agents have their local, very sophisticated models, and they share them to get even better models. In this case, non-IID is unavoidable. Here you can assume that you have all the bandwidth that you want, so you can do something more sophisticated.*

FedAvg works reasonably well in the first case, to get better rates and tackle the second case, we need something better. Let's see what.

Recall the problem that we want to solve and the federated learning simplifications,

$$\begin{aligned}\min_{\boldsymbol{x}\in\mathbf{R}^n} \quad & \mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)] \\ & \equiv \frac{1}{C}\sum_{c=1}^{C}\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)] \approx \frac{1}{C}\sum_{c=1}^{C}\mathbf{E}_{\theta\sim\Theta_c}[f(\boldsymbol{x};\theta)] \\ & \approx \frac{1}{C}\sum_{c=1}^{P}\frac{1}{N_c}\sum_{j=1}^{N_c} f_j(\boldsymbol{x}) = \frac{1}{C}\sum_{c=1}^{C} f_c(\boldsymbol{x}).\end{aligned}$$

Now, the fundamental issue is that the optimizer that we want to find is different from the one that we can find,

$$\begin{aligned}\boldsymbol{x}^* &= \arg\min_{\boldsymbol{x}\in\mathbf{R}^n}\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)] = \frac{1}{C}\sum_{c=1}^{C}\arg\min_{\boldsymbol{x}\in\mathbf{R}^n}\mathbf{E}_{\theta\sim\Theta}[f(\boldsymbol{x};\theta)] \\ &\neq \frac{1}{C}\sum_{c=1}^{C}\arg\min_{\boldsymbol{x}\in\mathbf{R}^n}\mathbf{E}_{\theta\sim\Theta_c}[f(\boldsymbol{x};\theta)] = \frac{1}{C}\sum_{c=1}^{C}\boldsymbol{x}_c^*,\end{aligned}$$

so, no matter the mixing matrix $\mathbf{W}$ we select at the server level, the model weights $\boldsymbol{x}$ are going to drift from the optimal model ones. We call this phenomenon client-drift.

The questions you should ask yourself at this point are at least two: How can we fix it? Do we need to fix it?

As engineers, of course we would like to get better models, so we want to fix it. However, we need to be careful when to do it: sometimes the data distributions are so different than local models are the only thing you need.

Said so, we are going to see one way to fix the issue of non-IID data with variance reduction, which as we know is a type of gradient tracking. The method is called SCAFFOLD. Many other methods exist, but we will focus on this one due to its popularity. Since it is based on a type of gradient tracking, SCAFFOLD is a generalization of SAGA, as we will prove. And, incidentally, SCAFFOLD will have better convergence guarantees than FedAvg, and we will be able to attain the optimal linear rate.

7.4.2 SCAFFOLD: setting, assumptions, algorithm

Rewrite the closest problem we can solve, for C client as,

$$\min_{\boldsymbol{x}\in\mathbf{R}^n} f(\boldsymbol{x}) = \frac{1}{C}\sum_{c=1}^{C}\left(f_c(\boldsymbol{x}) := \mathbf{E}_{\theta\in\Theta_c}[f(\boldsymbol{x};\theta)]\right)$$

Call $\boldsymbol{g}_{i,c}(\boldsymbol{x}) := \nabla f(\boldsymbol{x};\theta_{i,c})$ as the gradient sampled from the training data i (or batch i) belonging to client c.

Assumption 7.1 *Assume that $\boldsymbol{g}_{i,c}(\boldsymbol{x})$ is an unbiased estimator for $\nabla f_c(\boldsymbol{x})$ with variance,*

$$\mathbf{E}_i[\|\boldsymbol{g}_{i,c}(\boldsymbol{x}) - \nabla f_c(\boldsymbol{x})\|^2] \leqslant \pi^2.$$

Cross check: e.g., for SAGA, we have access to f_c directly, so $\pi = 0$. Rem for SGD/SAGA our cost is the empirical risk,

$$\min_{\boldsymbol{x}} \frac{1}{C} \sum_{c=1}^{C} f_c(\boldsymbol{x})$$

where each "client" has some realizations of the data.

Here, we solve instead,

$$\min_{\boldsymbol{x}} \frac{1}{C} \sum_{c=1}^{C} \mathbf{E}_{\theta \in \Theta_c}[f(\boldsymbol{x};\theta)] \approx \min_{\boldsymbol{x}} \frac{1}{C} \sum_{c=1}^{C} \frac{1}{N_c} \sum_{i=1}^{N_c} f(\boldsymbol{x};\theta_i)$$

Now, back to SCAFFOLD: Assume also the following.

Assumption 7.2 *We have two constants $G \geqslant 0, B \geqslant 1$ such that:*

$$\frac{1}{C} \sum_{c=1}^{C} \|\nabla f_c(\boldsymbol{x})\|^2 \leqslant G^2 + B^2 \|\nabla f(\boldsymbol{x})\|^2, \quad \forall \boldsymbol{x}.$$

This extends the assumptions in the IID setting and the bound G assumption there.

Let us interpret the two assumptions. The bound π takes care of the inter-client variance, i.e., it quantifies how accurate the approximation:

$$f_c(\boldsymbol{x}) = \mathbf{E}_{\theta \in \Theta_c}[f(\boldsymbol{x};\theta)] \approx \frac{1}{N_c} \sum_{i=1}^{N_c} f(\boldsymbol{x};\theta_i)$$

is.

On the other hand, the constants G, B take care of the non-IIDness.

As typical in machine learning, many sets of different assumptions exist, but the bottom line remains the same: you try to bound the inter-client variance, and you try to bound the infra-client variance. Also here we talk about convex function, but ML and FL cares about non-convex function (more). Most of the results can be extended to smooth non-convex functions of various type. An important type is the weakly convex type.

We are now ready to present the SCAFFOLD algorithm.

Stochastic Controlled Averaging for Federated Learning (SCAFFOLD)

Server update.

Initialize $\boldsymbol{x}_0, \boldsymbol{y}_0, \boldsymbol{c}_c$. *For each time* $t = 1, \dots$ *do:*

- *Select* $P \leqslant C$ *clients, and for each client*
  - **Communication to client step:** *Send* $\boldsymbol{x}_t, \boldsymbol{y}_t$ *to client*
  - **Communication from client step:** *Receive* $\Delta\boldsymbol{x}^p_{t+1}, \Delta\boldsymbol{y}^p_{t+1}$ *from the* p*-th* **client update**
- **Computation step:** *Perform the mixing*

$$(\boldsymbol{x}_{t+1}, \boldsymbol{y}_{t+1}) = (\boldsymbol{x}_t + \gamma\frac{1}{P}\sum_{p=1}^{P}\Delta\boldsymbol{x}^p_{t+1}, \boldsymbol{y}_t + \frac{1}{C}\sum_{p=1}^{P}\Delta\boldsymbol{y}^p_{t+1})$$

Client update.

Start from $\boldsymbol{x}^i_0 = \boldsymbol{x}_t, \boldsymbol{y}^i_0 = \boldsymbol{y}_t$. *Select batches* $\mathcal{B}$ *of data, each containing* B *data points.*
For epoch $k = 1, \dots, E$

- *Select a data batch* $\mathcal{B}_j$ *and perform a local SAGA update as,*

$$\boldsymbol{x}^c_{k+1} = \boldsymbol{x}^c_k - \alpha\left(\boldsymbol{g}_{j,c}(\boldsymbol{x}^c_k) - \boldsymbol{c}_c + \boldsymbol{y}_t\right),$$

- *Compute new quantities,*

$$\boldsymbol{c}^+_c = \begin{cases} \boldsymbol{g}_{j,c}(\boldsymbol{x}_t) & \text{option I} \\ \boldsymbol{c}_c - \boldsymbol{y}_t + \frac{1}{E\alpha}(\boldsymbol{x}_t - \boldsymbol{x}^c_{E+1}) & \text{option II} \end{cases}$$

and the delta's:

$$(\Delta\boldsymbol{x}^c_{t+1}, \Delta\boldsymbol{y}^c_{t+1}) = (\boldsymbol{x}^c_{E+1} - \boldsymbol{x}_t, \boldsymbol{c}^+_c - \boldsymbol{c}_c) \qquad \text{then: } \boldsymbol{c}_c \leftarrow \boldsymbol{c}^+_c.$$

The algorithm is a extension of both SAGA in the federated setting and FedAvg in the non-IID setting. If you select, the parameter $\gamma = 1$, $\alpha = \alpha_k$ and do not consider $-\boldsymbol{c}_c + \boldsymbol{y}_t$, then you get back FedAvg. In SCAFFOLD, we use two learning rates α, γ, which can be constant.

7.4.3 SCAFFOLD: convergence guarantees

Let us now look at the convergence properties of the algorithm. We have the following theorem.

Theorem 7.3 (SCAFFOLD convergence) *For any* L*-smooth and* m*-strongly convex* f_c *function, with the assumptions 7.1-7.2, the output of SCAFFOLD has expected error smaller than* ϵ *for small enough step size selections* $\alpha \leqslant \frac{1}{L}$ *and iterations bounded as*

$$T = \tilde{O}\left(\frac{\pi^2}{mEP\epsilon} + \frac{L}{m} + \frac{C}{P}\right)$$

where $\tilde{O}(\cdot)$ *means* $O(\cdot)$ *up to poly-log factors.*

Proof. *See [KKM+20].* ♣

The theorem states that SCAFFOLD can obtain a $O(1/T)$ convergence for constant learning rate. This is still not optimal for this class of problems; however, this is truly do to π^2. When, as in SAGA, $\pi^2 = 0$, then $T = \tilde{O}(1)$ which matches SAGA $O(\log(1/\epsilon)) = \tilde{O}(1)$, and we get back the linear convergence one expects. The results are given, as in the referenced paper, in a rather cryptic fashion. We can also derive the more usual form,

$$\mathbf{E}[F(\bar{\boldsymbol{x}}_T) - F(\boldsymbol{x}^*)] = \tilde{O}\left(\frac{\pi^2}{mTCE} + \exp(-T)\right),$$

for a suitable average sequence $\bar{\boldsymbol{x}}_T$.

SCAFFOLD is not the only method for non-IID data that matches SAGA and the research stays very rich.

7.4.4 Numerical results: kernel example

We look back at your kernel regression example, now in the non-IID setting. In Figure 7.7, we report a comparison between FedAvg and SCAFFOLD for the full batch setting ($B = 20$). We select Option I for SCAFFOLD. We can see how for $E = 1$, SCAFFOLD reduces to SAGA. We also see how SCAFFOLD benefits from local computation (a larger E): reduces communication without compromising convergence. We know that this is not the case for FedAvg in the non-IID setting.

FedAvg on the left $\longrightarrow$ SCAFFOLD on the right

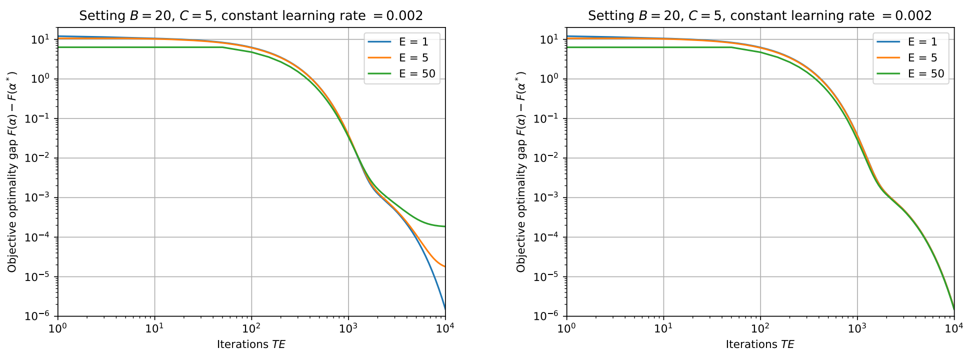


Figure 7.6. *A comparison between FedAvg (left) and SCAFFOLD (right).*

We then turn to not full client participation ($C = 3$). Once again, we compare FedAvg and SCAFFOLD. Here for SCAFFOLD, we select Option II.

FedAvg on the left $\longrightarrow$ SCAFFOLD on the right

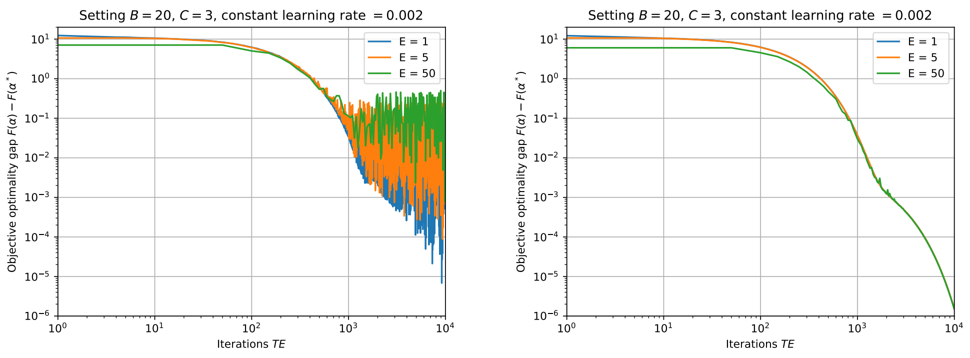


Figure 7.7. *A comparison between FedAvg (left) and SCAFFOLD (right).*

Finally, we look at the communication rounds to obtain a certain accuracy for SCAFFOLD, observing that the algorithm really benefits from $E \gg 1$.

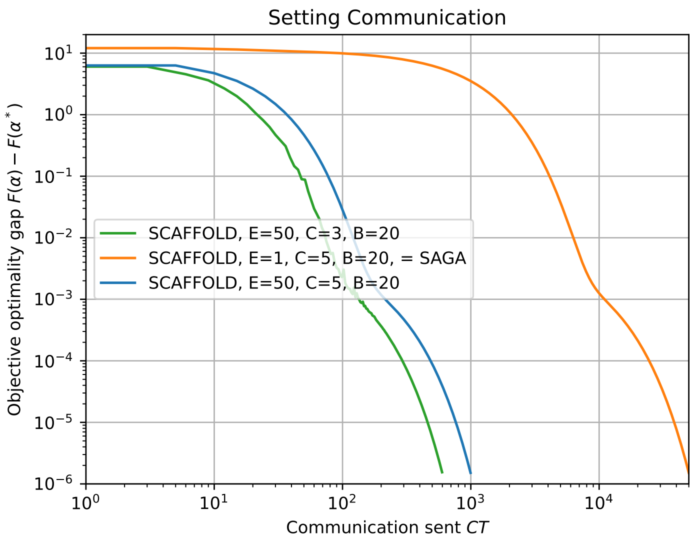


Figure 7.8. *Communication rounds to obtain a certain accuracy with SCAFFOLD.*

Chapter 8

Advanced federated learning

8.1 Introduction

In this chapter, we present two interesting recent developments in federated learning, we give an overview of others (by means of research papers), and we work out some exam questions. The material in this chapter (except for the exam questions) is really at the forefront of research in this domain, and we give only the general ideas.

8.2 Peer-to-peer federated learning

You may recall that the FedAvg algorithm can be written in an operator-theoretic fashion as,

$$\boldsymbol{y}_{t+1} = \mathbf{W}_t \left[\prod_{k=1}^{E} (I - \alpha_k \nabla F_k) \right] \circ \boldsymbol{y}_t, \tag{8.1}$$

, see the discussion around Equation 7.5, if you don't remember. Now, it makes sense to ask whether one could remove the need of a server node, by designing the weight matrix $\mathbf{W}_t$ appropriately.

In particular, instead of having an underlying star topology (all the clients communicating to the sever), you could define an arbitrary topology (clients talking to other clients). This is relatively easy to do in practice, and we have some foundations, back in Chapter 3 to help us with the theory.

Substituting the standard FedAvg "topology" with a distributed topology, mixes FedAvg with distributed optimization and consensus. As we have seen in Chapter 3 this carries more errors, but it is theoretically possible. Typically, the convergence guarantees in peer-to-peer FedAvg will depend on the spectral gap (with an additional term), and on the choice of the communication protocol. So the FL task is typically slowed down and you know who is the culprit.

Research paper 16 *Vogels, Thijs, Hadrien Hendrikx, and Martin Jaggi.* Beyond spectral gap: The role of the topology in decentralized learning. *Advances in Neural Information Processing Systems 35 (2022): 15039-15050.*

Research paper 17 *Chen, Mingzhe, Nir Shlezinger, H. Vincent Poor, Yonina C. Eldar, and Shuguang Cui.* Communication-efficient federated learning. *Proceedings of the National Academy of Sciences 118, no. 17 (2021): e2024789118.*

8.3 FL in the cross-silo setting: the role of Personalization

Let's now take a step back and look at the realistic cross-silo setting. In this case, you have a few entities with large quantities of data, which train their local models. Visualise it as training a classification model to detect cancer in a hospital setting. Each hospital is an agent/entity and hospitals may share their models to get better global models. Are they really better these aggregated models? And for whom are they better?

This question brings a bit beyond "classical" optimization into heuristic evidence. Recall what we wanted to solve and what we ended up solving:

$$\min_{\boldsymbol{x}} \mathbf{E}_{\theta\in\Theta}[f(\boldsymbol{x};\theta)] \approx \min_{\boldsymbol{x}} \frac{1}{P}\sum_{p=1}^{P} \mathbf{E}_{\theta\in\Theta_p}[f(\boldsymbol{x};\theta)] \approx \min_{\boldsymbol{x}} \frac{1}{P}\sum_{p=1}^{P}\frac{1}{N_p}\sum_{i=1}^{N_p} f_i(\boldsymbol{x}).$$

Especially in cross-silo FL, the original problem (the one with Θ) does not make sense: Θ_p are so different, that trying to get back to Θ has little meaning. So, maybe it is better to stay with the local models alone? Actually, that too may be too restrictive.

One of the leading idea is to add a *personalization* step to the training. In this line of thought, you share the model, do the mixing, but then "project" back onto the local dataset. How can one accomplish this?

A possible way to do this is by changing the local functions f_c, adding a local bonus. So instead of $f_c(\boldsymbol{x})$, you have $f_c(\xi_c(\boldsymbol{x}))$. And in particular you will have a local model ξ_c and a global model $\boldsymbol{x}$. In this way, you are decoupling the local and the global models so that they may converge to different quantities.

Possible examples of the local bonus are the following,

$$\text{(Regularization)} \qquad f_c(\xi_c(\boldsymbol{x})) = \min_{\xi_c}\{f_c(\xi_c) + \frac{\lambda}{2}\|\xi_c - \boldsymbol{x}\|^2\}\,;$$

$$\text{(Initialization)} \qquad f_i(\xi_c(\boldsymbol{x})) = f_i(\underbrace{\boldsymbol{x} - \alpha\nabla f_c(\boldsymbol{x})}_{=\xi_c})\,.$$

The first example tries to find a local model ξ_c which fits the data and it is not that far from the global model, by adding the penalization term $\|\xi_c - \boldsymbol{x}\|^2$. The second example defines the local model as the global one with an extra local gradient step. As you may realize, the possibilities here are multiple, and the research stays very active.

The rest stays the same, and you can use your FedAvg applied to $f_c(\xi_c(\boldsymbol{x}))$ to determine both ξ_c and $\boldsymbol{x}$.

Research paper 18 *Fallah, Alireza, Aryan Mokhtari, and Asuman Ozdaglar.* Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach. *Advances in neural information processing systems 33 (2020): 3557-3568*

8.4 Additional advanced algorithms and applications*

8.4.1 FedProx: an alternative of SCAFFOLD*

Research paper 19 *Li, Tian, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith.* Federated optimization in heterogeneous networks. *Proceedings of Machine learning and systems 2 (2020): 429-450.*

8.4.2 Newton-based federated learning: DANE*

Research paper 20 *Shamir, Ohad, Nati Srebro, and Tong Zhang.* Communication-efficient distributed optimization using an approximate newton-type method." In International conference on machine learning, pp. 1000-1008. PMLR, 2014.

8.4.3 Model-contrastive federated learning: MOON*

Research paper 21 *Li, Qinbin, Bingsheng He, and Dawn Song.* Model-contrastive federated learning. *In Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, pp. 10713-10722. 2021.*

8.4.4 Applications*

Research paper 22 *Xu, Jie, Benjamin S. Glicksberg, Chang Su, Peter Walker, Jiang Bian, and Fei Wang.* Federated learning for healthcare informatics. *Journal of healthcare informatics research 5 (2021): 1-19.*

8.5 Worked-out exam questions

Example 8.1 (Exam 2023, Language models) *Consider a federated learning setting, where a number of users use their phones to text messages. The phones record the messages to train a model to perform next-word prediction.*

Phones in nearby geographical locations are connected to the same server and employ the same model architecture. Let $\ell_p(x;\theta_i)$ be the loss incurred by phone p, when a message θ_i is revealed, and a next-word prediction x is used. As in class, $x \in \mathbf{R}^n$ are the model weights, or loosely "the model". The aim is to find the best x sharing information with the server.

Let the loss $\ell_p(x;\theta_i) : \mathbf{R}^n \times \mathbf{R}^m \to \mathbf{R}$ be strongly convex and smooth. The messages $\theta_i \in \mathbf{R}^m$ can be thought of as random vectors drawn from a certain probability distribution Θ_p.

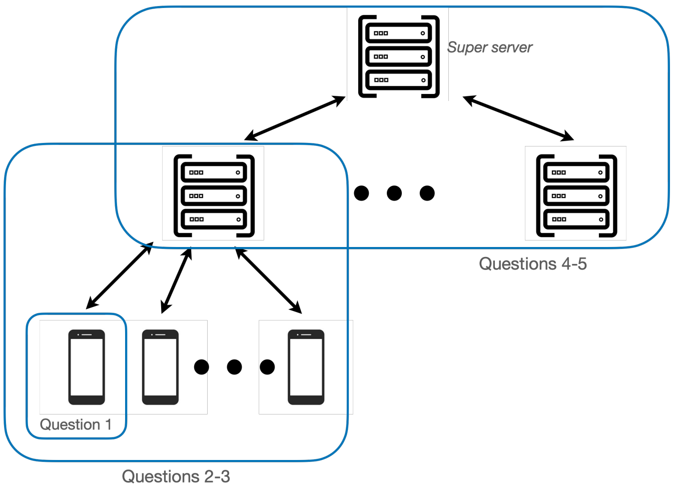


Figure 8.1. *A depiction of this problem and the different questions.*

1. *Consider the p phone alone. Describe the best stochastic method we have seen to solve the training problem,*
$$\min_{x\in\mathbf{R}^n} \mathbf{E}_{\theta_i\sim\Theta_p}[\ell(x;\theta_i)].$$
Detail the assumptions as well as the theoretical guarantees.

2. *Consider now the phones collaborating with one server. Describe the FedAvg algorithm that each phone could use to train its model (i.e., to learn the best x) by collaborating with the server. Detail the assumptions as well as the theoretical guarantees. Is this a cross-device or cross-silo setting?*

3. *The server is now collaborating to other servers to further improve the model. Other servers may be in other geographical locations and the probability distributions Θ_p may be very different. Describe a federated algorithm that can help the servers to collaborate to deliver the best model. In this setting, assume that you can have a* super-server *that coordinate all the servers.*

4. *Each server works on very special geographical locations and dialects. As such, a global model may not be well suited to its need. Describe a method to fine-tune (i.e., personalize) the global model for different servers.*

Sketched solutions

1. The best stochastic method we have seen in class for this class of problems is SAGA. For a strongly convex and smooth loss we have a linear convergence guarantee. You can describe here a bit more what this means.
2. This a cross-device setting and you can describe FedAvg.
3. We are now in a cross-silo setting. The best algorithm we know is SCAFFOLD. I can describe it here.
4. Here I can describe the personalized federated learning algorithm.

Chapter 9

Privacy issues

9.1 Introduction

In this chapter, we will (very) briefly study privacy issues in cooperative optimization and learning. It goes without saying that maintaining private data private is key in many applications, however it is often not trivial to achieve. Here, we will talk about three possible methods to achieve privacy, and the last one is the gold standard in optimization and learning.

Before beginning, it is important to ask oneself the question of what privacy means. What is private data for us? Private data can mean different things in different settings. For us, in some applications, we would like to keep the local cost functions f_i (which are generated by the data) private, in some other applications, we would like to keep the local features or models $\boldsymbol{x}^i$ private. The way we approach the problem will depend on our notion of privacy.

Here, we will look at the three methods: (1) data anonymization; (2) homomorphic encryption; and (3) differential privacy. I will not go deep into the details, but a good survey on the latter method is [DR14].

9.2 Anonymization

The first method we are going to look at is data anonymization. You may have heard about it: if you have some sensitive data about individuals, you may try to strip or delete the name of the individuals to preserve their privacy. If you only care about processing the data and not the names of the people involved, then this approach may work.

Often, we see this methodology used for health records and machine learning applications therein. For instance, if you want to train a neural network to predict breast cancer, you don't need the names of the individuals who provided the images.

Let us say that you are not curious and you do not care about the actual individuals names. Then, anonymization is probably all that you need. However, this is very different if you are curious and/or ill-intentioned.

In a very impactful 2013 paper [DMHVB13], researchers showed that, given the large amount of public information available for any given individual, one could reconstruct the deleted names with very high confidence in at least one application. In another application, other researchers were able to match health records to a specific high-profile individual.

What does this mean? It means that (1) we share too much information about ourselves online; (2) if someone wants one can associate anonymized and sensitive data with you, based on such public information. This is why some advertising company can profile you very well; this is why facebook's and google's data is very expensive to buy; that is why scammers are getting very good.

OK, but what does this mean *for cooperative optimization and learning*? It means that, in general, distributed optimization is not per se privacy-preserving.

9.2.1 Distributed optimization

We have discussed in several chapters in these lectures that one goal of cooperative optimization (and learning) is to keep data, i.e., the f_i's private. We said that we could achieve that by distributed algorithms. This is rather true, provided that *we are not curious, and we don't have access to public, side-information.* Let us see how to break privacy in the other case.

We will focus on a simple example. Consider a system with two devices, labelled 1 and 2, each equipped with their own private cost function $f_i(\boldsymbol{x})$. They are tasked to solve cooperatively the usual problem,

$$\min_{\boldsymbol{x}\in\mathbf{R}^n} \sum_{i=1}^{2} \left[f_i(\boldsymbol{x}) := \frac{1}{2}\|A_i\boldsymbol{x} - \boldsymbol{y}_i\|^2 \right],$$

where the A_i's are features matrices, $\boldsymbol{y}_i$ are the labels, and $\boldsymbol{x}$ is the global model weights. We need to make sure that device 1 does not have access to $\boldsymbol{y}_2$ and vice-versa.

Gradient Tracking. Consider the gradient tracking algorithm. There, you will never exchange $\boldsymbol{y}_i$ to your neighbor, only the gradient and the current model update $\boldsymbol{x}^i_k$. However, after a certain time k, we know (from theory) that we have approximately reached optimality, and therefore,

$$\nabla f_1(\boldsymbol{x}^*) + \nabla f_2(\boldsymbol{x}^*) \approx 0.$$

So, if you are device 1, you also know the value of $\nabla f_2(\boldsymbol{x}^*)$.

In addition, if you have side information on the fact that device 2 is training a similar model than you, i.e., $A_1 \approx A_2$ (e.g., the features are the same), then,

$$\nabla f_2(\boldsymbol{x}^*) \approx A_1^\top(A_1\boldsymbol{x}^* - \boldsymbol{y}_2),$$

so you can infer the private data $\boldsymbol{y}_2$.

Dual decomposition. Consider dual decomposition. The same trick works here too, since at each step we solve,

$$\boldsymbol{x}^i(\lambda_k) = \arg\min_{\boldsymbol{x}}\{f_i(\boldsymbol{x}) \pm \lambda_k^\top\boldsymbol{x}\} \iff \nabla f_i(\boldsymbol{x}^i) \pm \lambda_k = 0.$$

Since at each step you share $\boldsymbol{x}^i$ with your neighbor and you know λ_k, then you can infer $\nabla f_i(\boldsymbol{x}^i)$! And with side information $\boldsymbol{y}_i$: for example for device 1,

$$\nabla f_2(\boldsymbol{x}^2) - \lambda_k = 0 \implies A_2^\top(A_2\boldsymbol{x}^2 - \boldsymbol{y}_2) = +\lambda_k \approx A_1^\top(A_1\boldsymbol{x}^2 - \boldsymbol{y}_2).$$

So, simple distributed optimization ($\approx$ anonymization) is not in general enough for curious devices.

9.3 Encryption

The second method we look at is data encryption. You may have heard about it: you may render the data private by encrypting it, so that only yourself can read it, *but others can use it to do their computations.*

This may sound weird, but it does work for many applications, for instance your credit card data on the internet is encrypted.

Let us define an encryption function $\varphi(\boldsymbol{x})$, and we endow this encryption with rules, such that we can do simple sums and multiplications in the encrypted domain, as if we were doing them in the non-encrypted domain.

In particular, define such a function $\boldsymbol{x} \mapsto \varphi(\boldsymbol{x})$, such that,

$$\varphi(\boldsymbol{x}_1) \circ \varphi(\boldsymbol{x}_2) \equiv \boldsymbol{x}_1 \circ \boldsymbol{x}_2$$

for a given set of operations $\circ$ (e.g., multiplications, additions).

If such a function existed then, one could encode the private data $\boldsymbol{x}$, perform the operations needed publicly, and then decode the result. It is somewhat surprising, but functions like $\varphi(\boldsymbol{x})$ exists, and such an encryption goes under the name of **Homomorphic encryption** (with several standards for different operations).

For instance, the homomorphic encryption that code your credit card data is the RSA encryption protocol.

One of the major drawback for optimization and leaning is that homomorphic encryption makes vectors and matrices grow in size considerably, rendering the approach computationally heavy. If we already are in large-scale applications, growing the variable dimension is not something that would really work for us.

The interested reader can have a look at the survey [AAUC18] and a nice application in face recognition [EFG+09], but we won't pursue this method further here.

9.4 Differential privacy

The third method that we study in this chapter is differential privacy. This is the today's gold standard in many applications involving your data. The main digital players (Google, Microsoft, etc.) use it, and research on it is in full bloom.

As federated learning, also differential privacy was introduced by industry (Microsoft Research) collaborating with academia, this time in 2006. Two good introductions on the topic are the already cited book [DR14] and [CDH+16]. See also the very recent survey [CW24] for up-to-date references and results.

The key concept in differential privacy is signal-to-noise ratio: if there is a lot of data, you can afford to disclose your private data, provided you add enough noise that you make your data look like everybody else's data. In this context, you are *differentially* private: you are private with respect to the others, nobody can distinguish it is you. And in particular, your data will stay private against adversaries with arbitrary side information.

9.4.1 Intuition

Before going into the math, let us see how differential privacy works in practice. Let us work out a simple example.

Example 9.1 *Age is a sensitive information for most individual. Imagine you would like to compute the average age of a certain crowd at a concert, to inform the band about future advertising campaigns. Start with N people attending the concert. You want to know the mean of their age, you ask them, and they reply with age plus a zero-mean random noise. If the noise is big enough, nobody tells you their true age, but if N is big enough, the mean will be computed accurately! In fact,*

$$\mathit{result} = \frac{1}{N}\sum_{i=1}^{N}[\mathit{age}_i + r_i] \approx \mathbf{E}[\mathit{age} + r] = \overline{\mathit{age}}.$$

This example shows that one can compute simple operations and average out the noise. This is quite interesting, since in most distributed optimization and federated learning algorithms we are taking means of quantities. So differential privacy could be a good method for cooperative problems.

However, we will need to be careful. Differential privacy was originally designed for static (one-shot) databases. Cooperative algorithms are iterative: we compute the means (or mixing) many times. For example, recall the consensus iteration (3.3):

$$\boldsymbol{y}_{k+1} = \mathbf{W}\boldsymbol{y}_k, \qquad k = 1, \ldots, K.$$

The iterative process is the hard part for differential privacy.

Think about Example 9.1, but now ask the age over and over, and each time k the individuals reply with a different noise r_i, then, at each iteration k we can reconstruct the mean age:

$$\text{result}_k = \frac{1}{N}\sum_{i=1}^{N}[\text{age}_i + r_{i,k}] \approx \mathbf{E}_i[\text{age} + r_k] = \overline{\text{age}},$$

but if we store all the responses, we can also compute the individual age as,

$$\frac{1}{K}\sum_{k=1}^{K}[\text{age}_i + r_{i,k}] \approx \text{age}_i + \mathbf{E}_k[r_k] = \text{age}_i.$$

Since you may compute an average across the iterations, and the noise may disappear there too, and you can reveal the age of a single agent.

So, we need to study the noise process within the algorithmic framework a bit carefully. Typically, there is going to be a trade-off between noise level, N, and K.

9.4.2 Definition

We are now ready to mathematically define the concept of differential privacy. Consider two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, that are equivalent in all but one data.

The idea here is ask any query Q to the databases and see if we can distinguish them. If we can, then the one data difference can be identified and (possibly) the data itself can be revealed.

Definition 9.1 (ϵ differential privacy) *Given any two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, that are equivalent in all but one data. Consider any query Q asked to the databases. Then, one is ϵ differential private (ϵ-DP in short) if the probability distribution associated to query Q to database $\mathcal{D}_1$ is bounded as,*

$$\mathsf{P}(\mathcal{D}_1|Q) \leqslant \mathrm{e}^{\epsilon}\mathsf{P}(\mathcal{D}_2|Q)$$

for any query Q.

And therefore, by symmetry,

$$\mathsf{P}(\mathcal{D}_1|Q) \geqslant \mathrm{e}^{-\epsilon}\mathsf{P}(\mathcal{D}_2|Q).$$

The definition just means that the databases $\mathcal{D}_1$ and $\mathcal{D}_2$ are almost undistinguishable in probability up to multiplicative scalars.

9.4.3 Laplacian mechanism

Let us try to understand a bit better the definition. Let us consider the **Laplacian mechanism**.

Definition 9.2 (Laplacian mechanism) *We call Laplacian mechanism the act of adding a Laplacian distributed noise to our algorithms.*

We recall that the Laplacian noise distribution of mean 0 with scale b is given by,

$$\text{Lap}(x|b) = \frac{1}{2b}\mathrm{e}^{-\frac{|x|}{b}}.$$

Reconsider now the age example in Example 9.1 and ask the question: what is the uncertainty in the response that we must introduce in order to hide the participation of a single individual?

Imagine that is the first individual that is different across the two databases. Introduce the sensitivity

$$|\text{age}_1^1 - \text{age}_1^2| \leqslant \Delta f$$

for age of $_1$ individual and database ^{i}. We formalize the uncertainty as: the sensitivity of a function gives an upper bound on how much we must perturb its output to preserve privacy.

Now compute ϵ-DP when adding Laplacian noise.

We add noise to the respective ages age_1^1 and age_1^2 and the resulting noisy versions are x_1 and x_2. Now we look at the noise distribution, when asking if the age is z to both databases. By definition,

$$\frac{\mathsf{P}(\mathcal{D}_1|z)}{\mathsf{P}(\mathcal{D}_2|z)} = \frac{\exp(-|x_1 - z|/b)}{\exp(-|x_2 - z|/b)} = \exp([|x_2 - z| - |x_1 - z|]/b) \leqslant \exp\left(\frac{\Delta f}{b}\right).$$

If you then set the noise variance $b = \Delta f/\epsilon$, then the Laplacian mechanism is ϵ-DP. I.e., you cannot distinguish one individual contribution as long as it doesn't change the database of more than Δf.

This is the basic idea which is applied in real scenarios and databases, and in many flavors.

9.5 DGD with Laplacian noise

Let us see how to implement differential privacy in optimization. As we will see, adding noise to the messages we send, it is a easy business, proving convergence and privacy is another story.

First of all, let us define what we mean by privacy in distributed optimization.

Goal. Make sure that any algorithm operating on two sets of local cost functions equal in all but by one of them, will be undistinguishable in the DP sense.

This goal can very involved in theory, but in practice it is easy to put together, for example in decentralized gradient descent (from [WN23]):

DGD with Laplacian Noise (DGD-DP)

- *Start initializing at random $\boldsymbol{x}_k^i$ for all devices i, define a positive vanishing sequence $\{\gamma_k\} \to 0$, and a stepsize sequence $\{\alpha_k\}$*
- *Iterate for all $k = 0, 1, \ldots$*
    - **Obfuscate then communicate step:** *Add a Laplacian noise ζ_k^i to the local variable $\boldsymbol{x}_k^i$ as $\chi_k^i := \boldsymbol{x}_k^i + \zeta_k^i$, and send the obscured state χ_k^i to your neighbors*
    - **Computation step:** *Each device i updates the state*

$$\boldsymbol{x}_{k+1}^i = \boldsymbol{x}_k^i + \sum_{j \in N_i, j \neq i} \gamma_k \widehat{w}_{ij} (\chi_k^j - \boldsymbol{x}_k^i) - \alpha_k \nabla f_i(\boldsymbol{x}_k^i).$$

This algorithm is a version of DGD with some weighting γ_k to limit the noise level and variable stepsize.

9.5.1 Equivalence with DGD in the no-noise case

Consider now the following assumptions. First, we let $\widehat{w}_{ij} = \widehat{w}_{ji} \geqslant 0$ for $i \neq j$, and $\widehat{w}_{ii} = -\sum_{j \neq i} \widehat{w}_{ij}$. We also define the matrix $\widehat{W} = [\widehat{w}_{ij}]$, which is symmetric and $\widehat{W}\mathbf{1} = \mathbf{0}$. Assume that $W = \widehat{W} + I$ is doubly stochastic, then,

The update:

$$\boldsymbol{x}_{k+1}^i = \boldsymbol{x}_k^i + \sum_{j \in N_i, j \neq i} \gamma_k \widehat{w}_{ij} (\chi_k^j - \boldsymbol{x}_k^i) - \alpha_k \nabla f_i(\boldsymbol{x}_k^i)$$

is the standard DGD for $W = \widehat{W} + I$ doubly stochastic, no noise, and $\gamma_k = 1$, since we can write,

$$\boldsymbol{y}_{k+1} = \boldsymbol{y}_k + \widehat{\mathbf{W}}\boldsymbol{y}_k - \alpha_k \nabla F(\boldsymbol{y}_k) = \mathbf{W}\boldsymbol{y}_k - \alpha_k \nabla F(\boldsymbol{y}_k).$$

9.5.2 Theoretical guarantees

DGD-DP is an iterative noisy algorithm. Till very recently, no one could prove optimality and privacy. We recall that DGD with no-noise and vanishing stepsize enjoys a zero asymptotical error (See..). However here we are injecting noise in the updates.

As in the discussion ..., one of the main issue in proving optimality and privacy was that if we allow one to run the algorithm indefinitely, then one can also average on the iterative sequence to reveal the data. The idea was (and still is in some cases) to give an iteration budget not to pass a privacy budget. This can be formulated as defining an ϵ budget for the ϵ-DP and a total number of iterations T for the algorithm and impose that,

$$\epsilon \leqslant \sum_{k=1}^{T} \frac{\alpha_k}{\nu_k},$$

where ν_k is the variance of the Laplacian distribution. This means that the higher the ϵ, then the lowest is the privacy, but I can run my algorithm longer and the accuracy will be higher.

However, in 2022, we discovered that by increasing the variance of the Laplacian distribution ν_k and diminishing γ_k faster, then optimality and privacy could be enforced together. This was a key moment in differential privacy (albeit under some restrictive assumptions) and it is summarized in the following theorems.

Theorem 9.1 (Convergence) *Consider the convex optimization problem*

$$\operatorname*{minimize}_{\boldsymbol{x}\in\mathbf{R}^n} \sum_{i=1}^{n} f_i(\boldsymbol{x}),$$

and the DGD-DP algorithm to find one of its solutions $\boldsymbol{x}^*$ in a ϵ-DP fashion.

Assume Lipschitz continuous gradients for f_i's, $W = [w_{ij}]$ symmetric and $I+W$ doubly stochastic with $\|I + W - \frac{\mathbf{1}\mathbf{1}^\top}{N}\| < 1$.

Assume that the injected Laplacian noise is zero mean and with variance σ_k^i for which,

$$\sum_{k=0}^{\infty} \gamma_k^2 \max_i (\sigma_k^i)^2 < \infty.$$

Then, if $\sum_{k=0}^{\infty} \gamma_k = \infty$, $\sum_{k=0}^{\infty} \alpha_k = \infty$, and $\sum_{k=0}^{\infty} (\alpha_k^2)/\gamma_k < \infty$, we have

$$\|\boldsymbol{x}_k^i - \boldsymbol{x}^*\| \to 0 \qquad \textit{almost surely } \forall i.$$

Theorem 9.2 (Privacy) *Furthermore, assume bounded sensitivity $\|\nabla f_i(x)\|_1 \leqslant C$.*

Let

$$\bar{\epsilon} = \sum_{k=1}^{T} \frac{2C\alpha_k}{\nu_k}, \quad \nu_k = (\sigma_k^i)^2.$$

Then DGD-DP is $\epsilon \leqslant \bar{\epsilon}$-DP up to iteration T.

Moreover if $\sum_{k=1}^{T} \frac{\alpha_k}{\nu_k} < \infty$, then DGD-DP is $\epsilon \leqslant \bar{\epsilon}$-DP for all T's.

Proof. *See [WN23].* ♣

Example 9.2 *An example of allowed sequences for Theorems 9.1 and 9.2 to work is $\alpha_k = O(1/k)$, $\gamma_k = O(1/k^{0.9})$, $\nu_k \leqslant O(k^{0.3}/\epsilon$. In fact,*

$$\sum_{k=0}^{\infty} \gamma_k = \infty, \sum_{k=0}^{\infty} \alpha_k = \infty, \sum_{k=0}^{\infty} (\alpha_k^2)/\gamma_k = \sum_{k=0}^{\infty} \frac{k^{0.9}}{k^2} < \infty, \quad \sum_{k=1}^{T} \frac{\alpha_k}{\nu_k} = \sum_{k=1}^{T} \frac{\epsilon}{k^{1.3}} < \infty.$$

While the proof is quite involved, the theorems give you a way to ensure privacy and optimality at the same time, by tuning the stepsize and noise in a careful fashion. Different papers may have different assumptions. For instance, the bounded sensitivity in the form of $\|\nabla f_i(x)\|_1 \leqslant C$ may be a strong requirement in many cases, so then you may try to relax it. But this may incur again in the trade-off between privacy and accuracy, and so forth.

9.5.3 Numerical results (kernel setting)

We revisit our kernel setting in a ϵ-DP setting, where we decide to inject Laplacian noise to guarantee a certain level of privacy. In Figure 9.1, we report the convergence plots for DP-DGD for various level of ϵ and the parameters,

$$\alpha_k = \frac{0.002}{1 + 0.001k}, \gamma_k = \frac{1}{1 + 0.001k^{0.9}}, \nu_k = \frac{0.01}{\epsilon}\frac{1}{1 + 0.001k^{0.1}}.$$

Are the other assumptions of the theorems verified here?

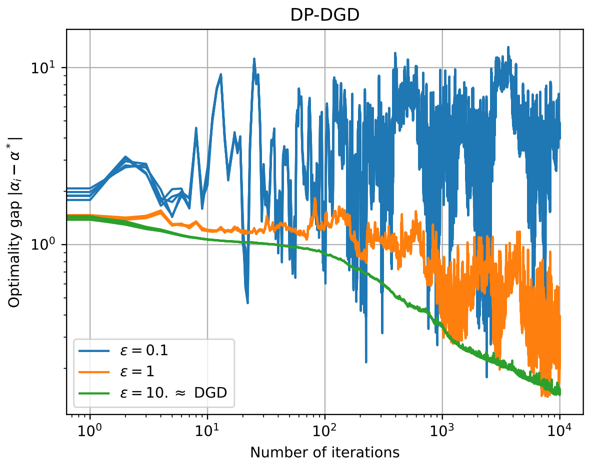


Figure 9.1. *Convergence plots for DP-DGD in the kernel example.*

9.6 Gradient Tracking with Laplacian noise

9.6.1 A tracking result

In [WN23], we can also find a gradient tracking version with Laplacian noise and similar results as before. I spare you the details.

But, take-home: Privacy is hard in distributed optimization. Optimization put hard requirements on what you can do, and adding noise makes you go very slowly to the optimizer. In this sense, DO is good for collaboration, not really for privacy.

9.6.2 Impossibility results under different assumptions*

Research paper 23 *Huang, Lingying, Junfeng Wu, Dawei Shi, Subhrakanti Dey, and Ling Shi.* Differential privacy in distributed optimization with gradient tracking. *IEEE Transactions on Automatic Control (2024).*

9.7 Differentially Private Federated Learning

As in distributed optimization, we may want to do federated learning in a differential private fashion, by adding noise to the model we exchange with the server. Once again, we will encounter the same trade-offs we have seen in optimization, either privacy or accuracy.

The literature in DP for FL is varied, in the sense that the way people do DP in FL depends on the models you use in FL. Since there are many ways to do federated learning, typically there are many approaches to DP for federated learning. For instance [WLD$^+$20, EOA22]. Here, we look mainly at [GKN17] and at the approach in [NBD22].

9.7.1 (ϵ, δ) differential privacy

First thing first, classical ϵ-DP is a bit too restrictive in general, and we relax it to (ϵ, δ)-differential privacy, by adding also an additive term.

Definition 9.3 ((ϵ, δ) differential privacy) *Given any two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$, that are equivalent in all but one data. Consider any query Q asked to the databases. Then, one is (ϵ, δ) differential private ((ϵ, δ)-DP in short) if the probability distribution associated to query Q to database $\mathcal{D}_1$ is bounded as,*

$$\mathsf{P}(\mathcal{D}_1|Q) \leqslant \mathrm{e}^{\epsilon}\mathsf{P}(\mathcal{D}_2|Q) + \delta$$

for any query Q.

This new definition does not change things dramatically, but in this way we can also use other noise distributions and we are less in trouble when probability vanishes.

9.7.2 Gaussian mechanism

We also introduce another "mechanism", the Gaussian noise, meaning we are adding Gaussian noise.

Definition 9.4 (Gaussian mechanism) *We call Gaussian mechanism the act of adding a Gaussian distributed noise to our algorithms.*

Example 9.3 (Regression) *Let us consider an example: the estimation of a real-valued function f. The Gaussian mechanism (sometimes GM in short) can approximate a real-valued function $f : D \to \mathbf{R}$ with a differentially private mechanism. Specifically, a GM adds Gaussian noise calibrated to the functions data set sensitivity S_f.*

This sensitivity is defined as the maximum of the absolute distance $\|f(d) - f(d')\|_2$, where d' and d are two adjacent inputs. The action of a GM is then defined as

$$M(d) = f(d) + \eta, \qquad \eta \sim \mathcal{N}(0, \sigma^2 S_f^2).$$

We can then bound the probability that (ϵ, δ)-DP is broken according to: $\delta \leqslant \frac{4}{5}\exp(-(\sigma\epsilon)^2/2)$, see [GKN17]. This means that, with a single query to the mechanism, δ needs to be $> \frac{4}{5}\exp(-(\sigma\epsilon)^2/2)$ to ensure (ϵ, δ)-DP.

9.7.3 DP FedAvg

Let us now introduce a possible way to render FedAvg (ϵ, δ)-differentially private. This is captured by the following algorithm.

DP - Federated averaging (DP-FedAvg)

```
Server update.
Initialize $\boldsymbol{x}_0$. For each time $t = 1, \ldots$ do:
  • Select $P \leqslant C$ clients, and for each client
      ◦ Communication to client step: Send $\boldsymbol{x}_t$ to client
      ◦ Communication from client step: Receive $\boldsymbol{x}^i_{t+1}$ from client update
  • Computation step: Perform the mixing
      $\boldsymbol{x}_{t+1} = \sum_{p=1}^{P} \frac{N_p}{\sum_{p=1}^{P} N_p} \boldsymbol{x}^p_{t+1}$

Client update.
Start from $\boldsymbol{x}^i_0 = \boldsymbol{x}_t$, Define the sensitivity $\mathcal{C}$ and the noise variance $\sigma_g^2$ as well as the scaling
$\beta = 2\mathcal{C}/B$. Select batches $\mathcal{B}$ of data, each containing $B$ data points.
For epoch $k = 1, \ldots, E$
  • Select a data batch $\mathcal{B}_j$ and compute the local gradient $g_{ij} = \nabla f_{\mathcal{B}_j}(\boldsymbol{x}^i_k)$
  • Clip the local gradient: $\tilde{g}_{ij} = g_{ij}/\max\{1, \|g_{ij}\|_2/\mathcal{C}\}$
  • Compute
      $\boldsymbol{x}^i_{k+1} = \boldsymbol{x}^i_k - \alpha_k(\tilde{g}_{ij} + \beta\eta), \qquad \eta \sim \mathcal{N}(0, \sigma_g^2)$
```

In the algorithm, you can notice the noise injection and the clipping of the gradient, to maintain the gradient within the sensitivity of the model. With this in place, we have the following theorem.

Theorem 9.3 (DP-FedAvg convergence) *Consider the same setting as in Theorem 7.1, with the addition of a clipping constant for which,*

$$\|\nabla f_c(\boldsymbol{x})\| \leqslant \mathcal{C}, \qquad \forall \boldsymbol{x},$$

and Assumption 7.2. Then DP-FedAvg can be made converge as,

$$\mathbb{E}[F(\bar{x}_T)] - F^* \leqslant \underbrace{O(\sqrt{\log(T/\delta)}/\epsilon)}_{privacy\ bound} + \underbrace{O(1/\sqrt{T}) + O(1/T)}_{optimization\ bound},$$

where $\bar{x}_T$ is an appropriate mean of the iterates.

Proof. *See [NBD22].* ♣

The theorem tells you that DP-FedAvg can obtain an error bound *which increases* as $\log(T)$, due to the privacy restriction. You can then appreciate the trade-off between having a good accuracy and a good privacy, once more.

9.7.4 DP-SCAFFOLD*

Research paper 24 *Noble, Maxence, Aurélien Bellet, and Aymeric Dieuleveut.* Differentially private federated learning on heterogeneous data. *In International Conference on Artificial Intelligence and Statistics, pp. 10110-10145. PMLR, 2022.*

Bibliography

[AAUC18] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[BPC⁺11] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *Foundations and Trends® in Machine Learning*, 3(1):1 – 122, 2011.

[BT97] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Athena Scientific, Belmont, Massachusetts, 1997.

[BXM03] Stephen Boyd, Lin Xiao, and Almir Mutapcic. Subgradient methods. *lecture notes of EE392o, Stanford University, Autumn Quarter*, 2004(01), 2003.

[CDH⁺16] Jorge Cortés, Geir E Dullerud, Shuo Han, Jerome Le Ny, Sayan Mitra, and George J Pappas. Differential privacy in control and network systems. In *2016 IEEE 55th Conference on Decision and Control (CDC)*, pages 4252–4272. IEEE, 2016.

[CW24] Ziqin Chen and Yongqiang Wang. Privacy-preserving distributed optimization and learning. *arXiv preprint arXiv:2403.00157*, 2024.

[DBLJ14] Aaron Defazio, Francis Bach, and Simon Lacoste-Julien. Saga: A fast incremental gradient method with support for non-strongly convex composite objectives. *Advances in neural information processing systems*, 27, 2014.

[DGBSX12] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(1), 2012.

[DMHVB13] Yves-Alexandre De Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 3(1):1–5, 2013.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[EFG⁺09] Zekeriya Erkin, Martin Franz, Jorge Guajardo, Stefan Katzenbeisser, Inald Lagendijk, and Tomas Toft. Privacy-preserving face recognition. In *Privacy Enhancing Technologies: 9th International Symposium, PETS 2009, Seattle, WA, USA, August 5-7, 2009. Proceedings 9*, pages 235–253. Springer, 2009.

[EOA22] Ahmed El Ouadrhiri and Ahmed Abdelhadi. Differential privacy for deep and federated learning: A survey. *IEEE access*, 10:22359–22380, 2022.

[FCT⁺20] F. Farina, A. Camisa, A. Testa, I. Notarnicola, and G. Notarstefano. `disropt` python package. *https://disropt.readthedocs.io/en/latest/index.html*, 2020.

[GG23] Guillaume Garrigos and Robert M Gower. Handbook of convergence theorems for (stochastic) gradient methods. *arXiv preprint arXiv:2301.11235*, 2023.

[GKN17] Robin C Geyer, Tassilo Klein, and Moin Nabi. Differentially private federated learning: A client level perspective. *arXiv preprint arXiv:1712.07557*, 2017.

[GL13] Saeed Ghadimi and Guanghui Lan. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM journal on optimization*, 23(4):2341–2368, 2013.

[GLQ⁺19] Robert Mansel Gower, Nicolas Loizou, Xun Qian, Alibek Sailanbayev, Egor Shulgin, and Peter Richtárik. Sgd: General analysis and improved rates. In *International conference on machine learning*, pages 5200–5209. PMLR, 2019.

[GOP19] M. Gürbüzbalaban, A. Ozdaglar, and P. A. Parrilo. Convergence rate of incremental gradient and incremental newton methods. *SIAM Journal on Optimization*, 29(4):2542–2565, 2019.

[HJ12] Roger A Horn and Charles R Johnson. *Matrix analysis.* Cambridge university press, 2012.

[KKM$^+$20] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International conference on machine learning*, pages 5132–5143. PMLR, 2020.

[KM$^+$21] Peter Kairouz, H Brendan McMahan, et al. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[LHY$^+$20] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. On the convergence of fedavg on non-iid data. In *International Conference on Learning Representations*, 2020.

[NBD22] Maxence Noble, Aurélien Bellet, and Aymeric Dieuleveut. Differentially private federated learning on heterogeneous data. In *International Conference on Artificial Intelligence and Statistics*, pages 10110–10145. PMLR, 2022.

[Nes04] Y. Nesterov. *Introductory Lectures on Convex Optimization.* Applied Optimization. Springer, 2004.

[NNC19] Giuseppe Notarstefano, Ivano Notarnicola, and Andrea Camisa. Distributed optimization for smart cyber-physical networks. *Foundations and Trends® in Systems and Control*, 7(3):253–383, 2019.

[NO09] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54(1):48–61, 2009.

[NOR18] Angelia Nedić, Alex Olshevsky, and Michael G Rabbat. Network topology and communication-computation tradeoffs in decentralized optimization. *Proceedings of the IEEE*, 106(5):953–976, 2018.

[RB16] E. K. Ryu and S. Boyd. Primer on Monotone Operator Methods. *Applied Computational Mathematics*, 15(1):3 – 43, 2016.

[SCJ18] Sebastian U Stich, Jean-Baptiste Cordonnier, and Martin Jaggi. Sparsified sgd with memory. *Advances in neural information processing systems*, 31, 2018.

[Sti18] Sebastian U Stich. Local sgd converges fast and communicates little. *arXiv preprint arXiv:1805.09767*, 2018.

[WLD$^+$20] Kang Wei, Jun Li, Ming Ding, Chuan Ma, Howard H Yang, Farhad Farokhi, Shi Jin, Tony QS Quek, and H Vincent Poor. Federated learning with differential privacy: Algorithms and performance analysis. *IEEE transactions on information forensics and security*, 15:3454–3469, 2020.

[WN23] Yongqiang Wang and Angelia Nedić. Tailoring gradient methods for differentially private distributed optimization. *IEEE Transactions on Automatic Control*, 69(2):872–887, 2023.

[XBL06] Lin Xiao, Stephen Boyd, and Sanjay Lall. Distributed average consensus with time-varying metropolis weights. *Automatica*, 1:1–4, 2006.

[YLY16] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.